Stream Algorithms and Architecture

by

Henry Hoffmann

B.S., University of North Carolina at Chapel Hill 1999

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2003

Author . . . .
Department of Electrical Engineering and Computer Science
May 22, 2003

Certified by . . . .
Anant Agarwal
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Stream Algorithms and Architecture

by

Henry Hoffmann

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2003, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

This work proposes ***stream algorithms*** along with a ***decoupled systolic architecture*** (DSA). Stream algorithms executed on a decoupled systolic architecture are notable for two reasons. The floating-point efficiency of stream algorithms increases as the number of processing elements in the DSA increases. Furthermore, the compute efficiency approaches 100 % asymptotically, that is for large numbers of processors and an appropriate problem size.

This thesis presents a methodology for structuring algorithms as stream algorithms, a virtual computer architecture for efficiently executing stream algorithms, and results gathered on the Raw[34] microprocessor that confirm that stream algorithms achieve high performance in practice.

Thesis Supervisor: Anant Agarwal
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

First and foremost, the author would like to thank Volker Strumpen. Volker has been instrumental in the development of the work presented in this thesis, and co-authored the earliest publication of these ideas [15]. In addition to his technical help, he has provided encouragement and a great deal of assistance improving the author's writing. It is safe to assume that this work would be far less mature and far more difficult to read if not for Volker's influence.

The author also thanks his advisor, Anant Agarwal, for supporting this work and for encouraging the implementation of stream algorithms on Raw, which greatly strengthened this thesis.

Special thanks go to Janice McMahon, Jeremy Kepner, and Bob Bond at MIT Lincoln Laboratory for their support and encouragement.

Thanks also to friends and family who put up with the author's moodiness and bizarre schedule during the past two years.

This work was made possible by the financial support of Lincoln Laboratory's Lincoln Scholars Program, DARPA, the NSF and the Oxygen Alliance.

Contents

1 Introduction **19**

1.1 Contributions 25

1.2 Road Map 26

2 Related Work **29**

2.1 Architecture and Algorithms 29

2.2 Stream Processing 31

3 A Decoupled Systolic Architecture **33**

4 Stream Algorithms **37**

4.1 Specifying Stream Algorithms 42

5 Matrix Multiplication **45**

5.1 Partitioning 45

5.2 Decoupling 46

5.3 Efficiency Analysis 48

6 Triangular Solver **51**

6.1 Partitioning 52

6.2 Decoupling 53

6.3 Efficiency Analysis 55

7 LU Factorization **59**

7.1 Partitioning 59

7.2 Decoupling 60

7.3 Efficiency Analysis 63

8 QR Factorization **69**

8.1 Partitioning 71

8.2 Decoupling 73

8.3 Efficiency Analysis 90

9 Convolution **97**

9.1 Partitioning 97

9.2 Decoupling 98

9.3 Efficiency Analysis 101

10 Implementation on Raw **103**

10.1 Matrix Multiplication 108

10.2 Triangular Solver 111

10.3 LU Factorization 114

10.4 QR Factorization 115

10.5 Convolution 120

10.6 Summary 122

11 Conclusion **125**

List of Figures

1-1 A processor designed to efficiently compute the inner product $c = a \cdot b$. The values of a and b are not stored in memory, but are transmitted over the network. 21

1-2 A systolic array for matrix multiplication. 21

1-3 An architecture supporting a decoupled systolic matrix multiplication. Compute processors implement a systolic matrix multiplication, while memory processors along the periphery of the compute array are responsible for storing data that will be used in later phases of computation. 24

1-4 The bar graph shows the ratio of the speed of a distributed memory implementation of a matrix multiplication to that of a stream algorithm implementation. All matrices are of size $N \times N$. 25

3-1 A decoupled systolic architecture (DSA) is an $R \times R$ array of compute processors (P) surrounded by $4R$ memory processors (M) as shown for $R = 8$. Compute processors use fast memory in form of a register set only. Each memory processor consists of a compute processor plus a memory interface to a slow memory of large capacity. 33

3-2 A compute processor contains a general-purpose register set (GPR), an integer unit (IU), and a floating-point unit (FPU) based on a multiply-and-add module. The processor connects via FIFO's to its four neighbors. 34

5-1 Seven time steps of a systolic matrix multiplication $C = A \cdot B$ for 2×2 matrices. Each box represents a compute processor. Values entering, leaving, or being generated in the array are shown in bold face. Shaded boxes mark the completion of an inner product. The data flow of the operands and products is split into the top and bottom rows. The pseudo code executed by each processor in the systolic array is shown in Figure 5-2. 46

5-2 Pseudo code for compute processor p_{ij} executing systolic matrix multiplication. One should note that this algorithm requires only a bounded amount of storage to hold the values x, n, i, j, R, and N. 47

5-3 Data flow of a compute-efficient matrix multiplication $C = A \cdot B$ for 4×4 matrices on 2×2 compute processors. Shaded boxes on the periphery mark memory processors, and indicate the completion of an inner-product otherwise. 49

6-1 Systolic lower-triangular solver for $N = 3$ and two right-hand sides. The pseudocode executed by each processor in the systolic array is shown in Figure 6-2. 54

6-2 Pseudo code for compute processor p_{ij} executing systolic lower triangular solver. One should note that each processor requires a bounded amount of storage to hold the values x, n, j, R. 55

6-3 Phases of a decoupled systolic lower-triangular solver on an $R \times R$ array of compute processors. In phase 1, Equation 6.2 is solved for X_{11}. In phase 2, B_{21} is updated according to Equation 6.4. While the matrix multiplication is executed on the compute processors, the matrix subtraction is performed on the memory processors as they receive each individual result of $A_{21}X_{11}$. Finally, in phase 3, Equation 6.6 is solved for X_{21}. The shapes of the matrix areas indicate how the rows and columns enter the compute array in a staggered fashion. 56

7-1 Systolic LU factorization for $N = 3$. Figure 7-2 shows the pseudocode executed by each compute processor performing a systolic LU factorization. 61

7-2 Pseudo code for compute processor p_{ij} executing systolic LU factorization. Note that the storage of values i, j, x, and n requires a bounded amount of storage only. 62

7-3 Phases of stream-structured LU factorization on an $R \times R$ array of compute processors, with $N = 2R$. Phase 1 factors A_{11} into L_{11} and U_{11} according to Equation 7.2. Phase 2 solves Equation 7.3 for U_{12}. Phase 3 solves Equation 7.4 for L_{21}. Phase 4 computes A'_{22} according to Equation 7.5, performing the matrix subtraction on the memory tiles. Finally, in phase 5 A'_{22} is factored according to Equation 7.6. The shapes of the matrices indicate that the rows and columns enter the compute array in a staggered fashion. 64

8-1 The product of premultiplication $G(i,j)^T \cdot A$ differs from A in rows i and j only, while the product of postmultiplication $A \cdot G(i,j)$ differs from A in columns i and j only. 73

8-2 Systolic Givens computation for an $M \times R$ matrix, where $M \geq R$. The figure illustrates the triangularization of a 4×3 matrix A, such that $R = D^{-1/2}\ G(4,3)^T\ G(4,2)^T\ G(3,2)^T\ G(4,1)^T\ G(3,1)^T\ G(2,1)^T \cdot A$. The fast Givens transformation $G(i,j)$ is represented by the pair $(\alpha_{ij}, \beta_{ij})$, and $\delta_i = d_i^{-1/2}$. Figure 8-4 shows the pseudocode executed by each processor in the systolic array. [continued in Figure 8-3] 75

8-3 Continuation of Figure 8-2. 76

8-4 Pseudo code for compute processor p_{ij} executing systolic Givens computation. Here the code is shown for the phase that ouputs intermediate values of d to the bottom of the array. One should note that each processor requires only a bounded amount of storage to hold the values x, d, a, m, α, β, γ, δ, n, i, j, R, and M. [continued in Figure 8-5] . . 77

8-5 Continuation of Figure 8-4. [continued in Figure 8-6] 78

8-6 Continuation of Figure 8-5. 79

8-7 Systolic premultiplication for computing R and updating A according to Equation 8.4. Figure 8-9 shows the pseudocode executed by each processor in the array. [continued in Figure 8-8] 82

8-8 Continuation of Figure 8-7. 83

8-9 Pseudocode for compute processor p_{ij} executing systolic premultiplication. Note that each processor requires only a bounded amound of storage to hold the values of m, i, j, a, x, and M. 84

8-10 Systolic postmultiplication for computing an $R \times M$ block of $Q = IG(2,1)G(3,1)G(4,1)G(3,2)G(4,2)G(4,3)D^{-1/2}$. [continued in Figure 8-11] 86

8-11 Continuation of Figure 8-10. 87

8-12 Phases of a stream-structured QR factorization on an $R \times R$ array of compute processors, with $M = 3R$ and $N = 2R$. In phase 1, one triangularizes the left-most $N/2$ columns of A, producing R_{11} and a set of fast Givens transformations as expressed in Equation 8.3. In phase 2, the Givens transformations are applied to the right-most $N/2$ columns of A yielding R_{12}, A'_{22}, and A'_{32} according to Equation 8.4. In phase 3, one triangularizes the right-most columns of A according to Equation 8.5. Phases 4-6 compute Q_1 according to Equation 8.6, while phases 7-12 compute $Q = Q_1 Q_2$ according to Equation 8.8. Note that phases 3 and 7–12 can be modified without loss of efficiency, so that matrix Q is not distributed across the top and bottom rows of memory processors, but would be available on one side only. 89

8-13 (a) The fast Givens computation of Equation 8.3 (or Equation 8.5) for column block i effects the bottom $(\sigma_M - i + 1)R$ rows. (b) After triangularizing column block i, one updates the $(\sigma_M - i + 1)R \times (\sigma_N - i)R$ submatrix of A according to Equation 8.4. 91

8-14 One computes column block i of $M \times M$ matrix Q by partitioning the computation into σ_M row blocks, each with R rows and $(\sigma_M - i + 1)R$ columns. Each row block is computed using a systolic postmultiplication. After the postmultiplication, the hatched area of the matrix holds the final values of Q, while the shaded area comprises intermediate values. 93

9-1 Systolic convolution of input sequence a_i of length $M = 5$ with $N = 4$ weights w_j. Both the weights and input sequence are fed into the linear array of $R = 4$ compute processors. Intermediate results are shown above the corresponding processors. Value b_k^i represents an intermediate value of b_k after the first i products have been computed according to Equation 9.1. 99

9-2 Pseudo code for compute processor p_i in systolic convolution. 100

9-3 Stream convolution of an input sequence of length $M = 5$ with $N = 4$ weights on a linear array of $R = N/2 = 2$ compute processors and $M = 2$ memory processors. Value $b_k^{l,i}$ represents the computation of b_k when the outer summation of Equation 9.2 has been executed l times and the inner summation has been executed i times. Note that the memory tile on the right performs an addition to accumulate the results of the partial convolutions. 101

10-1 The efficiency of the compute processors executing a stream-structured matrix multiplication is shown as a function of N. The solid curve represents the results predicted by the efficiency analysis for an ideal DSA, while dotted curve represents measured results on Raw. 111

10-2 The efficiency of all processors (both memory and compute) executing a stream-structured matrix multiplication is shown as a function of $\sigma = N/R$, where $R = 4$. The solid curve represents the results predicted by the efficiency analysis for an ideal DSA, while the dotted curve represents physical results measured on Raw. 112

10-3 The efficiency of the compute processors executing a stream-structured lower triangular solver is shown as a function of N. The solid curve represents the results predicted by the efficiency analysis for an ideal DSA, while the dotted curve represents physical results using Raw to implement a DSA. 114

10-4 The efficiency of all processors executing a stream-structured lower triangular solver is shown as a function of $\sigma = N/R$, where $R = 4$. The solid curve represents the results predicted by the efficiency analysis for an ideal DSA, while the dotted curve represents physical results measured on Raw. 115

10-5 The efficiency of the compute processors executing a stream-structured LU factorization is shown as a function of N. The solid curve represents the results predicted by the efficiency analysis, while the dotted curve represents physical results measured on Raw. 116

10-6 The efficiency of all processors executing a stream-structured LU factorization is shown as a function of $\sigma = N/R$, where $R = 4$. The solid curve represents the results predicted by the efficiency analysis for an ideal DSA, while the dotted curve represents physical results measured on Raw. 117

10-7 The efficiency of the compute processors executing a stream-structured QR factorization and computing only R is shown as a function of N. The solid curve represents the results predicted by the efficiency analysis for an ideal DSA, while the dotted curve represents physical results measured on Raw. 118

10-8 The efficiency of all processors executing a stream-structured QR factorization, but omitting the computation of Q is shown as a function of $\sigma = N/R$, where $R = 4$. The solid curve represents the results predicted by the efficiency analysis, while the dashed curve represents physical results using Raw to implement a DSA. 119

10-9 The efficiency of the compute processors executing a stream-structured QR factorization is shown as a function of N. The solid curve represents the results predicted by the efficiency analysis, while the dotted curve represents results measured on Raw. 120

10-10The efficiency of the all processors executing a stream-structured QR factorization is shown as a function of $\sigma = N/R$, where $R = 4$. The solid curve represents the results predicted by the efficiency analysis for an ideal DSA, while the dotted curve represents physical results measured on Raw. 121

10-11The efficiency of the compute processors executing a stream-structured convolution is shown as a function of the length N of the shorter vector. The solid curve represents the results predicted by the efficiency analysis for an ideal DSA, while the dotted curves represents physical results measured on Raw for three separate data sizes. 122

10-12The efficiency of all processors executing a stream-structured convolution is shown as a function of $\sigma = N/R$, where $R = 4$ and N is the length of the shorter vector. The solid curve represents the results predicted by the efficiency analysis, while the dotted curves represents physical results for three different data sizes using Raw to implement a DSA. 123

List of Tables

11.1 Summary of stream algorithms. The table shows the number of compute processors P and the number of memory processors M. In addition, the table compares the execution time T and compute efficiency E and compares that to the maximum compute processor efficiency achieved on a 4×4 configuration of Raw tiles, E_{raw}. 126

Chapter 1

Introduction

Stream algorithms and the decoupled systolic architecture represent a holistic approach to algorithm and computer architecture design that together provide a scalable way to achieve efficient computation on a general purpose programmable computer. This thesis shows that many important applications can be restructured as stream algorithms and executed on a decoupled systolic architecture. Furthermore, the floating-point efficiency of these algorithms increases as the number of processors used to execute the algorithm increases.

This approach, combining algorithm and architecture development, is best illustrated through the simple example of the multiplication of two $N \times N$ matrices. Matrix multiplication is usually represented by a standard triply-nested loop, the core of which is an inner-product:

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    for (k=0, c[i][j]=0.0; k<N; k++)
      C[i][j] += A[i][k]*B[k][j];
```

Assuming that `N` is very large, these loops can be unrolled to effectively reduce the loop overhead. Therefore the main source of inefficiency in this algorithm is the memory operations required to load the values of `A` and `B` and to store the values of `C`. If the processor on which this code is executed has a multiply-and-add instruction, then the inner loop would execute two loads (one for the value of A and one for the value of B) and a single multiply-and-add instruction for every loop iteration. Thus, for a single issue processor, the maximum efficiency of this loop would be 33 %, ignoring all other overhead (including cache misses and store instructions). Therefore, to increase the efficiency of such an algorithm, one must consider methods of eliminating inefficient memory operations from the critical path of the computation.

One method to eliminate the loads and stores associated with an inner product on a single processor is to allow functional units to operate directly on data from the network rather than from memory. Such an architecture must expose network accesses to the programmer in the processor's instruction set[1]. Assuming that such a processor is connected to a neighbor on the north, south, east, and west and its instruction set contains a floating-point multiply-and-add[2] instruction, one can rewrite the innermost loop of the matrix multiplication as:

```
for (k=0, c=0.0; k<N; k++)
  fma c, c, $W1, $N2
```

Here it is assumed that values of a arrive from the west on the network port represented by $W1, values of b arrive from the north on the network port represented by $N2, and finally c is stored in a local register (c.f. Figure 1-1). The ability to operate directly on data coming from the networks allows programmers to eliminate memory operations. Therefore, one expects this version of the inner-product to run at least three times faster than the initial version. However, one would still like to exploit the parallelism of the matrix multiply and utilize additional processors in the computation.

To efficiently incorporate other processors into the computation the instruction set must expose network routing to the programmer. Such an interface allows pro-

[1]Exposing network resources to the programmer is a key design aspect of the Raw architecture [34]. In fact, the networks in Raw are tightly integrated into the processor pipeline. This is a design feature which earlier register-mapped network architectures including the Connection Machine CM2 [13] and Warp [2] lack.

[2]A floating-point multiply-and-add instruction or fma is defined such that after execution of fma d, c, a, b; d has the value of c+a*b.

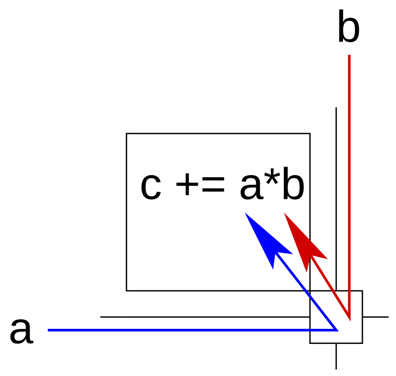


Figure 1-1: A processor designed to efficiently compute the inner product $c = a \cdot b$. The values of a and b are not stored in memory, but are transmitted over the network.

grammers to completely overlap communication and computation. Thus, one may parallelize the matrix multiplication by rewriting the inner-most loop as:

```
for (k=0, c=0.0; k<N; k++)
  fma c, c, $W1, $N2    route $W1->$E1, $N2->$S2
```

and replicating this loop across multiple processors. This pseudo-assembly code performs an `fma` instruction while simultaneously routing data from the west to the east and from the north to the south. Again, such code has no memory operations and efficiently incorporates many processors into the computation (c.f. Figure 1-2).

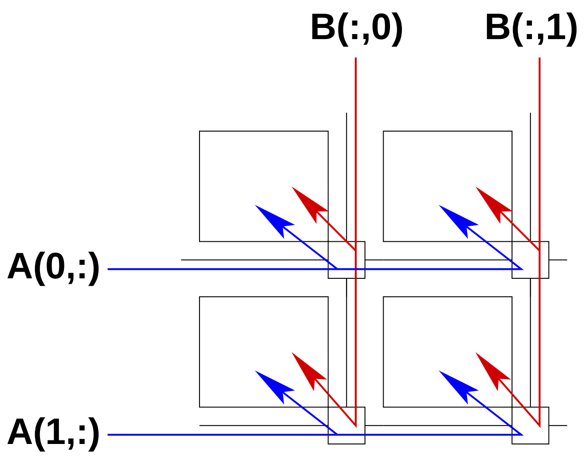


Figure 1-2: A systolic array for matrix multiplication.

The architecture described thus far is a programmable systolic array, and therefore has similar benefits and drawbacks to a systolic array [21]. Users can write highly efficient programs for such an architecture so long as the number of available processors matches the problem size. For example, an $N \times N$ matrix multiplication requires an $N \times N$ array of processing elements. Thus, systolic arrays are highly efficient, but also highly specialized for a single problem size. This specialization is at odds with the goal of a general purpose programmable architecture.

A commonly accepted technique for overcoming the specialization of systolic arrays is the ***simulation*** of multiple processing elements on a single systolic processor. Such a processor uses local memory to store intermediate values of computation. If the architecture is designed to support arbitrarily large problems, the amount of local memory required is unbounded, and thus the local memory available on such a processor must be large. Large memories require a load/store interface to handle intrinsically long access times. Thus, simulation of a systolic array reintroduces the original source of inefficiency: memory operations. Therefore, simulation is inadequate for highly efficient computation and other methods must be found to achieve the benefits of systolic arrays but maintain the flexibility to compute arbitrarily large problems on a general purpose architecture.

Out-of-core algorithms provide insight into how to divide up a large problem efficiently among limited resources [37]. The goal of out-of-core algorithms is to efficiently schedule computation on problems that are too large to fit into the main memory of a particular processor. These techniques provide the inspiration to ***partition*** a large problem into systolic phases, each of which can be solved without simulation given the resources at hand. For example, suppose one is presented with an $R \times R$ array of compute resources and an $N \times N$ matrix multiplication where $N > R$. One could continuously apply the block-partitioning of Equation 1.1 until each of the block matrices in the matrix multiplication can be computed systolically on an $R \times R$ array of processing elements.

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \tag{1.1}$$

This technique allows one to apply systolic methods within a general purpose framework without the inefficiency of simulation. However, using this partitioning requires that one store those parts of the matrices that are not needed during a given systolic phase of computation. Furthermore, the data must be accessed in a manner that does not reintroduce memory operations to the critical path of computation.

Decoupled access execute architectures provide inspiration to completely separate computation from memory accesses [30]. Data values only need to be accessed on the edges of those processors that implement the systolic array. In this manner, processors on the edges of the array are responsible for continuously accessing data and sending it to those processors performing the actual computation. Therefore, the systolic array of *compute processors* is augmented with *memory processors*. Memory processors are given access to a large, but slow, memory while compute processors have no access to local memory apart from a small, fast register set. Therefore, all load/store operations are performed on memory processors, which send data into the systolic array implemented by the compute processors. If one makes use of asymptotically fewer memory processors than compute processors, the operations performed on the memory processors become insignificant as the total number of processors grows. An architecture consisting of an array of compute processors augmented with a set of memory processors on the periphery is referred to as a ***decoupled systolic architecture***.

By applying this ***decoupling*** technique to the problem of matrix multiplication, one can keep an $R \times R$ array of compute processors fully utilized using only $2R$ memory processors. As shown in Figure 1-3, memory processors along the top of the compute array are responsible for storing columns of one operand matrix, while memory processors along the left side of the compute array are responsible for storing rows of the other operand.

This decoupled systolic matrix multiplication represents a highly efficient computation on a general purpose decoupled systolic architecture. While the memory processors contribute no useful computation, the compute processors implement highly efficient matrix multiplications. In the process of decoupling the matrix multiplication, one requires asymptotically fewer memory processors than compute processors.

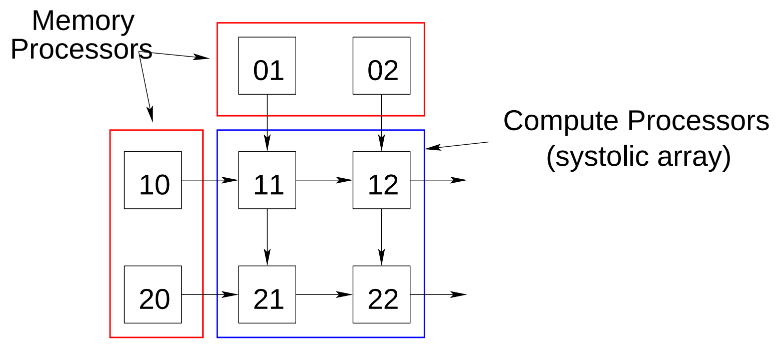


Figure 1-3: An architecture supporting a decoupled systolic matrix multiplication. Compute processors implement a systolic matrix multiplication, while memory processors along the periphery of the compute array are responsible for storing data that will be used in later phases of computation.

Thus, as the total number of processors available increases, those executing memory operations become insignificant. Therefore, the efficiency of such a matrix multiplication increases as the number of processors increases.

To confirm the benefits of stream algorithms, one can compare the performance of a stream-structured matrix multiplication to that of a distributed memory matrix multiplication. The Raw microprocessor supports both paradigms of computation, and thus serves as an ideal platform for this experiment. Both a stream algorithm and a distributed memory algorithm for matrix multiplication have been implemented on the Raw microprocessor. Figure 1-4 shows the ratio of the run time of the distributed memory implementation to that of the stream algorithm implementation. When the matrices are all square and of size 16×16, the stream matrix multiplication is 7.25 times faster than the distributed memory implementation. For problems of size 256×256, the stream version executes almost 16 times faster than the distributed memory version. These results show the dramatic decrease in run-time that can be achieved by using highly efficient stream algorithms[3]. The stream algorithms execute so much faster than the distributed memory operations because stream algorithms eliminate not only memory operations, but also cache misses.

The techniques presented for improving the performance of matrix multiplication

[3]The methodology for gathering these results and the implementation of stream algorithms on Raw is discussed in greater detail in Chapter 10.

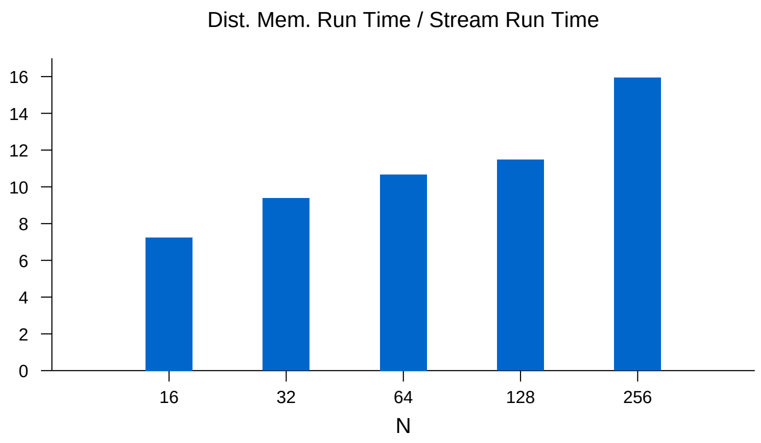


Figure 1-4: The bar graph shows the ratio of the speed of a distributed memory implementation of a matrix multiplication to that of a stream algorithm implementation. All matrices are of size $N \times N$.

can be generalized to serve as a framework for structuring many other parallel algorithms in a highly efficient manner. The term ***stream algorithm*** is used to denote an algorithm that has been partitioned into systolic phases and whose memory accesses have been decoupled from this systolic computation. When executed on a decoupled systolic architecture, stream algorithms can achieve 100 % floating point efficiency.

1.1 Contributions

The primary contribution of this thesis is the definition of a class of parallel algorithms named *stream algorithms* and an abstract architecture referred to as a *decoupled systolic architecture*, or *stream architecture*. Stream algorithms executed on a stream architecture have several unique features:

- Stream algorithms achieve very high floating point efficiencies, and therefore extremely fast run-times.
- In contrast to conventional wisdom, the efficiency of stream algorithms *increases* as the number of available compute resources increases. Furthermore, the compute efficiency of stream algorithms approaches 100 % asymptotically.

- Stream algorithms represent an excellent match for today's microarchitectures, which exploit fast, but short wires.

- Stream algorithms can be implemented on a general purpose architecture and will execute efficiently if that architecture implements the features of a decoupled systolic architecture.

In addition to defining stream algorithms, this thesis makes the following contributions:

- Stream algorithms are presented for several important problems, including QR factorization, which has been traditionally difficult to parallelize.

- The real-world applicability of stream algorithms is demonstrated by evaluating their performance on the Raw microprocessor, and it is shown that stream algorithms can achieve high efficiencies in practice.

- Issues of concern for the practical implementation of decoupled systolic architectures are discussed.

Hopefully this work will serve as a starting point for future development of parallel algorithms and architectures.

1.2 Road Map

The remainder of this thesis develops a methodology for developing architectures and algorithms that support a streaming manner of computation. The remainder of this work is organized as follows. Chapter 2 discusses related work in this area. In Chapter 3 the decoupled systolic architecture is discussed. Chapter 4 formally defines the notion of a stream algorithm, and the methodology for structuring algorithms as stream algorithms. Then, five examples are presented: a matrix multiplication in Chapter 5, a triangular system solver in Chapter 6, an LU factorization in Chapter 7, a QR Factorization in Chapter 8, and a convolution in Chapter 9. For all examples, it is shown how each problem can be reformulated using the stream structuring methodology and that the resulting stream algorithm achieves 100 % efficiency asymptotically when executed on a decoupled systolic architecture. In Chapter 10 all five examples are implemented on the Raw microprocessor, and the results are compared to what one might expect from an ideal DSA.

Chapter 2

Related Work

This chapter describes the relationship of the work in this thesis to previous work in architecture and algorithms and also in stream processing.

2.1 Architecture and Algorithms

The amount of transistors that fit onto a single chip has been growing steadily, and computer architects are reaching the critical mass for realizing a general-purpose parallel microarchitecture on a single chip. Research prototypes such as Trips [28], Smart Memories [25], and Raw [34] represent the first steps into the design space of ***tiled architectures***, which are single-chip parallel machines whose architecture is primarily determined by the propagation delay of signals across wires [14].

To enable high clock frequencies on large chip areas, tiled architectures have short wires that span a fraction of the side length of a chip, and use registers to pipeline the signal propagation. Short wires, in turn, introduce a scheduling problem in space and time to cope with the propagation of signals across distances longer than those reachable via a single wire. Moving data across wires and distributing operations across processors are equally important scheduling goals. This scheduling problem has received attention in the context of VLSI design [27], parallel computation [22], and parallelizing compiler design [38] in the past.

The speed advantage of short wires has not gone unnoticed. In fact, systolic arrays were proposed by Kung and Leiserson in the late 1970's [21, 20], and aimed, in part, at exploiting the speed of short wires. Lacking the chip area to support programmable structures, however, early systolic arrays were designed as special-purpose circuits for a particular application, and were customized for a given problem size.

Later systolic systems such as Warp [2] and iWarp [7] became programmable, so they could reap the benefits of systolic arrays without sacrificing flexibility. The Warp architecture has three principal components, a linear processor array composed of Warp cells, an interface unit (IU) and a host. Each Warp cell contains a floating point multiplier and a separate floating point adder, as well as several FIFOs that are used to communicate with neighbors. The host is a general purpose computer that provides data to the processor array via the interface unit. The IU is responsible for generating addresses and loop control signals for the processor array. The individual Warp cells do not contain integer units, although they do contain large local memories. Therefore, the interface unit is given the responsibility of generating address values and communicating these to the processor array. The Warp architecture is very similar to the decoupled systolic architecture presented in Chapter 3, with only two major differences:

1. Warp and iWarp decouple address generation, but not memory access itself, from computation.

2. The FIFOs used by Warp cells are not integrated into the functional unit bypasses. Thus, moving data from the FIFO to the register set requires an extra step on Warp as opposed to a decoupled systolic architecture.

The designers of Warp and iWarp note that "memory access is typically a bottleneck to performance" and thus they recommend systolic communication patterns in Warp programs [7]. The use of the local memory on each cell was further encouraged by the Warp compiler [12], which adopted a single-program, multiple-data view of the machine, and specifically scheduled memory accesses on all Warp cells. However, putting large local memories on each processing cell promotes inefficient simulation of systolic arrays for large problem sizes.

However, the significant area and energy efficiency of systolic arrays merit their reexamination in face of the architectural similarities to recent tiled microarchitectures. To fully realize these benefits one must not resort to the simulation of systolic arrays. An alternative, motivated by Decoupled Access/Execute Architectures [30] is decoupling memory accesses entirely from computation.

The main feature of a decoupled access/execute machine is that the architecture forces programs to be separated into two instruction streams that communicate via architectural queues. One instruction stream is responsible for accessing operands, while the other is responsible for executing operations on data. The decoupling concept has even been applied to stream processing and vector machines [5]. However, with today's microtechnology providing an abundance of resources on a single chip, it is possible to take this idea one step further and have multiple memory access units feeding multiple execute units in parallel.

Nonetheless, the architectural solution of using decoupling rather than simulation does not inform one of a method to efficiently execute relatively large problems on a relatively small programmable array. The problem is similar conceptually to that faced by the designers of out-of-core algorithms [37]. Out-of-core algorithms were designed to efficiently perform computations on data sets that are too large to fit into main memory and must be stored on disks. This class of algorithms provides insight into scheduling data accesses to maximize data reuse. The same insights suggest ways to schedule the computation of a large problem on a small programmable systolic array.

2.2 Stream Processing

The notion of a stream abstract data type has been around for decades [1]. Recently this subject has gained a great deal of attention as data streams present a useful abstraction for a number of important applications such as digital signal processing, multimedia processing, and network processing. Compiler writers [31, 36, 18], computer architects [19], and algorithm designers [3, 4] have all worked to exploit the stream data type for performance gains. This previous work shares two central design principles with the work presented in this thesis: consider architecture and algorithms simultaneously and schedule operations to reduce the overhead of memory accesses.

There are, however, some major differences between previous work in stream programming and the work presented here on stream algorithms. Stream programming has been traditionally based on the observation that many important problem domains naturally contain a stream abstraction. Programming systems are then built that exploit this abstraction. Not surprisingly, this work has the shown that the performance of stream programming systems on streaming applications exceeds that of a more general purpose approach.

The work of stream algorithms, on the other hand, is based on developing a systematic and principled approach to achieving scalable and efficient computation on a general purpose architecture. Rather than focus on mapping streams into language and hardware constructs, stream algorithms work to schedule instructions such that operands are constantly streaming through functional units. Furthermore, stream algorithms are evaluated on whether or not they achieve 100 % efficiency asymptotically. Where the focus of previous stream programming systems was on manipulating the stream abstraction, the focus of stream algorithms is on efficient general purpose computation.

The major thrust of this thesis is to present stream algorithms, which are based on a unique combination of the ideas of systolic arrays, decoupled access/execute architectures, and out-of-core algorithms. This work presents a principled methodology for algorithm design and a general purpose computing architecture that together achieve high efficiency and scalable performance.

Chapter 3

A Decoupled Systolic Architecture

A decoupled systolic architecture (***DSA***) is a set of characteristics that describe a single-chip tiled architecture. In this sense, a DSA represents a virtual machine or an abstract architecture. Any tiled architecture that implements all the characteristic features described in this section can be referred to as a DSA.

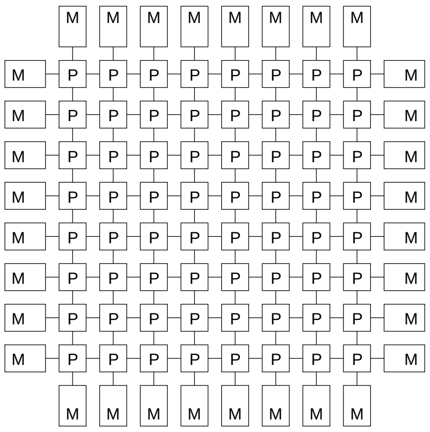


Figure 3-1: A decoupled systolic architecture (DSA) is an $R \times R$ array of compute processors (P) surrounded by $4R$ memory processors (M) as shown for $R = 8$. Compute processors use fast memory in form of a register set only. Each memory processor consists of a compute processor plus a memory interface to a slow memory of large capacity.

A DSA is a set of processors connected in a mesh topology by a fast network of short wires as shown in Figure 3-1. The DSA consists of an $R \times R$ array of ***compute processors***, augmented with $4R$ ***memory processors*** on the periphery of the compute array. The peripheral memory processors are the distinguishing feature of a

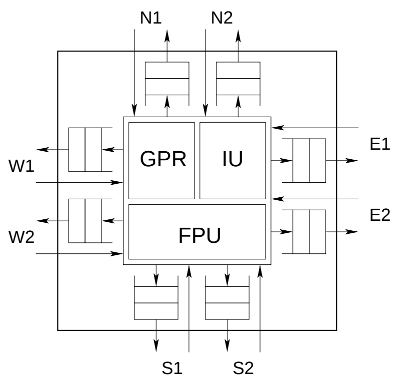


Figure 3-2: A compute processor contains a general-purpose register set (GPR), an integer unit (IU), and a floating-point unit (FPU) based on a multiply-and-add module. The processor connects via FIFO's to its four neighbors.

DSA. Each of the memory processors consists of a compute processor with access to an additional large memory. It is assumed that the memory can deliver a throughput of two loads or one load and one store per clock cycle. The compute processors can be implemented in one of many architectural styles with varying degrees of efficiency, for example, VLIW, TTA, or superscalar. However, the choices for achieving 100 % compute efficiency in an area-efficient fashion are more limited. As the DSA represents an abstract architecture, this section focuses on the key architectural features for a DSA without dwelling on the details of a particular instantiation.

The compute processor, shown in Figure 3-2, is a simple general-purpose programmable core comprising an integer unit, a floating-point unit with a multiply-and-add module as the centerpiece, and a multi-ported, general-purpose register set. The compute processor does not include a large local memory because of the intrinsic physical constraint that large memories have larger latencies than small ones. Instead, it has only a small but fast memory in the form of a register set. This lack of local memory also serves to reduce the foot print of a compute processor and thus allows more compute processors to be placed on a single chip. To focus attention on the datapath, Figure 3-2 omits all of the control logic and a small instruction memory. Each compute processor contains a single-issue, in-order pipelined FPU that allows issue of one floating point multiply-and-add operation per clock cycle.

In addition to the floating point multiply-and-add unit, each compute processor contains a high performance floating-point divide unit. Although, the number of divisions in all examples is asymptotically insignificant compared to the number of multiply-and-add operations, the latency of the divider can have a dramatic effect on efficiency for smaller problem sizes. To reduce the severity of this effect, the latency of the divider should be as small as possible and ideally, it should be fully pipelined[1].

Arguably the most important feature of a DSA is the design of its on-chip network. This interconnect uses two prominent features of Raw's static network [34]. The network is ***register-mapped***, that is instructions access the network via register names, and it is a ***programmed routing network*** permitting any globally orchestrated communication pattern on the network topology. The latter is important for stream algorithms that change patterns between the phases of the computation. These features are discussed in more detail in the following paragraphs.

As illustrated in Figure 3-2, each compute processor uses blocking FIFO's to connect and synchronize with neighboring processors. These FIFO's are exposed to the programmer by mapping them into register names in the instruction set. The outgoing ports are mapped to write-only registers with the semantics of a FIFO-push operation, and the incoming ports as read-only registers with the semantics of a FIFO-pop operation. Furthermore, it is preferable fro the network to be tightly integrated with the pipelined functional units. Accordingly, bypass wires that commonly feed signals back to the operand registers also connect the individual pipeline stages to the outgoing network FIFO's. The tight network integration ensures low-latency communication between neighboring compute processors, and allows for efficient pipelining of results from operations with different pipeline depths through the processor array.

The decoupled systolic architecture uses a wide instruction word to schedule multiple, simultaneous data movements across the network, between the functional units and the network, as well as between the register set and the network. A typical DSA instruction such as

```
fma $4,$4,$N1,$W2  route $N1->$S1, $W2->$E2
```

[1]A floating point divider that meets these requirements is detailed in [24].

consists of two parts. The `fma` operation is a floating-point multiply-and-add compound instruction. It multiplies the values arriving on network ports `N1` and `W2`, and adds the product to the value in general-purpose register `$4`. Simultaneously, it routes the incoming values to the neighboring processors as specified by the `route` part of the instruction. The value arriving at port `N1` is routed to outgoing port `S1`, and the value arriving at port `W2` to outgoing port `E2`. Instructions of the decoupled systolic architecture block until all network operands are available. Using small FIFO's with a length larger than one eases the problem of scheduling instructions substantially. There exists a trade-off between the instruction width and the area occupied by the corresponding wires within a processor. For the DSA, it is assumed that a maximum of three data networks and five move instructions can be specified within the route part of a single instruction.

Chapter 4

Stream Algorithms

In this section decoupled systolic algorithms, nicknamed stream algorithms[1], are presented along with a set of conditions for which one can increase efficiency by increasing the number of processors such that the compute efficiency approaches 100 %. Alternatively, one may view stream algorithms as the product of a program-structuring methodology. There are three design principles that characterize stream algorithms:

1. Stream algorithms are structured by ***partitioning*** a large problem into subproblems and solving the subproblems systolically. ***Systolic designs*** are well suited for parallel machines with a local interconnect structure and match the DSA with its fast but short wires.

2. Stream algorithms ***decouple memory accesses from computation*** by dedicating processors to one of the two tasks. This idea is motivated by the Decoupled Access/Execute Architecture [30].

3. Stream algorithms use M memory processors and P compute processors, such that the number of memory processors M is asymptotically smaller than the number of compute processors P, that is $\boldsymbol{M=o(P)}$.

[1]The name is derived from the fact that when executing at 100 % efficiency, all functional units must receive a continuous stream of inputs. Such a notion of a stream is consistent with the colloquial sense. According to Webster [26], a stream is "an unbroken flow (as of gas or particles of matter)," a "steady succession (as of words or events)," a "constantly renewed supply," or "a continuous moving procession (a stream of traffic)."

By conforming to these design principles, stream algorithms abandon load/store operations on compute tiles, and thus reduce the instruction count on the critical path.

The key strategy for the design of an efficient decoupled systolic algorithm is to recognize that the number of memory processors must be negligible compared to the number of compute processors, because memory processors do not contribute any useful computation. While it is often impossible to design an efficient decoupled systolic algorithm for a very small number of processors and a very small problem size, one can actually increase the efficiency for larger numbers of processors and large problem sizes. This observation is emphasized by formulating the decoupling-efficiency condition.

Definition 1 ***(Decoupling-Efficiency Condition)***
Given a decoupled algorithm with problem size N and a network of size R,[2] *let $P(R)$ be the number of compute processors and $M(R)$ the number of memory processors. Then, an algorithm is* ***decoupling efficient*** *if and only if*

$$M(R) = o(P(R)).$$

Informally, decoupling efficiency expresses the requirement that the number of memory processors becomes insignificant relative to the number of compute processors as one increases the network size R. Decoupling efficiency is a necessary condition to amortize the lack of useful computation performed by the memory processors. For example, suppose one implements an algorithm on $P = R^2$ compute processors. If one can arrange the memory processors such that their number becomes negligible compared to P when increasing the network size R, the resulting algorithm is decoupling efficient. Thus, for a decoupling-efficient algorithm with $P = \Theta(R^2)$, one may choose M to be $\Theta(\lg R)$, or $M = \Theta(R)$, or $M = \Theta(R \lg R)$. In contrast, a design with $M = \Theta(R^2)$ would not be efficiently decoupled. Decoupled

[2]The network size R is used as a canonical network parameter. The number of processing nodes is determined by the network topology. For example, a 1-dimensional network of size R contains R processing nodes, whereas a 2-dimensional mesh network contains R^2 processing nodes.

systolic algorithms per se are independent of a particular architecture. Note, however, that the DSA shown in Figure 3-1 is particularly well suited for executing either one such algorithm with $(P, M) = (\Theta(R^2), \Theta(R))$ or multiple algorithms concurrently with $(P, M) = (\Theta(R), \Theta(1))$.

Decoupling efficiency is a necessary but not sufficient condition to guarantee high performance. One determines the compute efficiency of a stream algorithm with problem size N on a network of size R from the number of useful compute operations $C(N)$, the number of time steps $T(N, R)$, and the area counted in number of processors $P(R) + M(R)$:

$$E(N, R) = \frac{C(N)}{T(N, R) \cdot (P(R) + M(R))}. \tag{4.1}$$

The product of time steps and area can be interpreted as the compute capacity of the DSA during time period T. For all practical purposes, one may relate the problem size N and network size R via a real-valued σ such that $N = \sigma R$. Substituting σR for N in Equation 4.1, compute efficiency is defined by means of the following condition.

Definition 2 ***(Compute-Efficiency Condition)***
An algorithm with problem size N is ***compute efficient*** *when executed on a network of size R, if and only if*

$$\lim_{\sigma, R \to \infty} E(\sigma, R) = 1,$$

where $N = \sigma R$.

Equation 4.1 implies a necessary condition for obtaining a compute-efficient algorithm: either the number of memory processors $M = 0$ or the algorithm is decoupling efficient. If one operates a compute array without any memory processors it is a systolic array. If one is concerned with a general purpose architecture, rather than a highly specialized systolic case, compute efficiency implies decoupling efficiency as a prerequisite. Thus, with decoupling efficiency as necessary condition for achieving

100 % compute efficiency asymptotically, every compute-efficient stream algorithm is decoupling efficient, whereas the converse is not true. The compute-efficiency condition requires that both $R \to \infty$ and $\sigma = N/R \to \infty$. Thus, in practice stream algorithms require that $N \gg R$, which is a realistic assumption since using a very large network implies that one intends to solve a very large problem. For $\sigma = 1$, the problem size matches the network size, and one operates the network as a systolic array. Since decoupling-efficient stream algorithms use an asymptotically smaller number of memory processors than compute processors, one may view stream algorithms as a subset of systolic algorithms with a restricted number of inputs and outputs. Inversely, one may view a systolic algorithm as a special case of a stream algorithm that is distinguished by $\sigma = 1$ in $N = \sigma R$. The trade-off between N and R is discussed during the discussion of stream algorithms below.

Before presenting concrete examples of stream algorithms, a general ***stream-structuring methodology*** is outlined, which consists of three steps:

Partitioning: Given a problem with $\sigma > 1$ in $N = \sigma R$, that is the problem size N is larger than the network size R, one starts by partitioning the problem into smaller, independent subproblems. Each of the subproblems as well as the composition of their results must be suitable for parallelization by means of a systolic algorithm such that the compute processors access data in registers and on the network only. For simple data-parallel applications, the partitioning can be obvious immediately. For applications with more complicated data dependencies, recursive formulations and partitioning methods like those developed for out-of-core algorithms [37] can be helpful. To simplify the design of the systolic algorithm, *retiming* [23] may be used. It allows one to start the design with a semi-systolic algorithm, which can be transformed automatically into a systolic algorithm if one exists [22]. The design of a semi-systolic algorithm can be significantly easier than that of a systolic version, because it permits the use of *long wires* that extend beyond next-neighbor processors. Note that the design of stream algorithms themselves are not concerned with wire length or the physical constraints of wire delay[14]. Rather, these limits are concerns that must be taken into account for a practical implementation of stream algorithms and a decoupled systolic architecture.

Decoupling: The goal is to move the memory accesses off the critical path. To this end, one must decouple the computation such that the memory accesses occur on the memory processors and compute operations on the compute processors. For a systolic problem, the memory processors feed the input streams into the compute processors, and the decoupling procedure is almost trivial. However, the composition of several subproblems requires careful planning of the flow of intermediate data streams, such that the output streams of one systolic phase can become input streams of a subsequent phase without copying streams across memory processors. Occasionally, it may be beneficial to relax the strict dedication of memory processors to memory accesses, and compute portions of the composition of the subproblems, such as reductions, on the memory processors themselves. Therefore a fully-fledged compute processor is integrated into each memory processor of the DSA.

Efficiency Analysis: After partitioning and decoupling, one has designed a stream algorithm. To qualify as a compute-efficient stream algorithm, however, it is required that the compute-efficiency condition holds. Therefore, the choice of the number of memory processors must be asymptotically smaller than the number of compute processors, and one must show that $E(\sigma, R)$ approaches 1 for large values of R. Meeting the compute-efficiency condition requires that one schedules the subproblems for optimal pipelining on the compute array. Experience shows that one may need to iterate over the partitioning and decoupling steps until a compute-efficient solution is found.

One may emphasize the concept of a stream algorithm by considering what a stream algorithm is not. (1) A stream algorithm is not a collection of N tasks that is scheduled on $R < N$ processors, using time sharing and context switching to guarantee progress. Instead, a stream algorithm is a computation structured such that the schedule of individual tasks is determined by the order of elements in the data streams and is primarily organized by the memory processors. Also, (2) a stream algorithm does not simulate $\lceil N/R \rceil = \lceil \sigma \rceil$ processors of a systolic array on one compute processor. While simulation of a systolic array is a generally applicable method for executing a parallel algorithm of problem size N on $R < N$ processors, each of the R processors needs an unbounded amount of memory to store the state of each of the $\lceil \sigma \rceil$ subproblems, and consequently additional instructions must be executed to manage a large local memory. In contrast, stream algorithms avoid local memory accesses entirely by decoupling the computation from memory accesses, and moving the memory accesses off the critical path.

4.1 Specifying Stream Algorithms

The systolic algorithms executed on the compute processors in the following sections of this thesis are specified in pseudocode. The pseudocode used here is based primarily on that of Cormen, Leiserson, and Rivest [8], and conforms to the following conventions.

1. Programs for the compute processors are specified in a single-program, multiple-data or SPMD manner. Thus, a single program is executed on all compute processors.

2. Indentation indicates scope and block structure.

3. The **for** loop construct has the same interpretation as in the Turing programming language [16].

4. The conditional constructs **if**, **else if**, and **else** are all supported.

5. Registers are specified by letters. The letters i and j are reserved to hold the coordinates of the compute processor within the array. Register i holds the processor's row, while register j holds the column.

6. Registers M, N, and R are all reserved to hold the values of the problem size (M, N) and the network size (R) for a particular invocation.

7. The assignment operator is $\leftarrow$.

8. Standard mathematical operators are supported.

9. Network ports are directly addressable by the construct $\text{Net}(dir)$, where dir can take on the value *north*, *south*, *east*, or *west*. There are three networks in each direction. They are differentiated by appending a number to the direction. For example, $\text{Net}(north2)$ specifies the second network port on the north. Each network port can communicate one word in each direction simultaneously. The network ports are implemented as FIFO's and can be operated on as registers, with two exceptions. If the network port appears as an operand, its value is consumed. If the network port is assigned a value, that value is transferred to the processor in the corresponding direction. For example, the statement $x \leftarrow \text{Net}(north)$ assigns register x the value on the first of the processor's northern input networks. The statement $\text{Net}(south2) \leftarrow x$ moves the value in register x out of the processor, to the south, on its second network.

10. In addition to specifying assignment and mathematical operations of registers and network ports, a single instruction may execute up to five move instructions. Generally, these instructions will involve moving data from the network into registers, or routing data from one network input to a network output. The pseudocode uses the **route** keyword to denote the part of the instruction that contains all moves that do not include mathematical operations. Move operations are specified with the $\rightarrow$ operator. For example, the instruction
    $x \leftarrow x + \text{Net}(north) \times y$, **route** $\text{Net}(north) \rightarrow \text{Net}(south)$
    assigns register x the value of x plus the product of register y and the value on the first northern input network. Simultaneously the instruction routes the value from the first northern input network to the first southern output network.

11. All operations, including the move operations that are specified by the **route** instruction execute concurrently. Thus, the value in a register or on a network port may be routed to many different networks or registers simultaneously.

12. Instructions which only execute data movement may be specified by the construct

    nop **route** ...

    which will perform all the move operations specified by the route, but will not execute any other operations.

13. Operator precedence is the same as in the Turing language.

Chapter 5

Matrix Multiplication

As the first example of a stream algorithm, consider a dense matrix multiplication. Given two $N \times N$ matrices A and B, one wishes to compute the $N \times N$ matrix $C = AB$. Element c_{ij} in row i and column j of product matrix C is the inner product of row i of A and column j of B:

$$c_{ij} = \sum_{k=1}^{N} a_{ik} \cdot b_{kj}, \tag{5.1}$$

where $1 \leq i, j \leq N$.

5.1 Partitioning

One may use a block-recursive partitioning for the matrix multiplication, recursing along the rows of A and the columns of B:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \end{pmatrix}. \tag{5.2}$$

For each of the matrices C_{ij}, $C_{ij} = A_{i1}B_{1j}$ where A_{i1} is an $N/2 \times N$ matrix and B_{1j} an $N \times N/2$ matrix. Thus, the matrix multiplication can be partitioned into a homogeneous set of subproblems.

5.2 Decoupling

Each product element c_{ij} can be computed independently of all others by means of Equation 5.1. In addition, Equation 5.2 allows one to stream entire rows of A and entire columns of B through the compute processors. Furthermore, one must partition a problem of size $N \times N$ until the C_{ij} are of size $R \times R$ and fit into the array of compute processors. One then implements the resulting subproblems as systolic matrix multiplications, illustrated in Figure 5-1 for $N = R = 2$. Figure 5-2 shows the pseudocode executed by each processor performing the systolic matrix multiplication. Rows of A flow from the left to the right, and columns of B from the top to the bottom of the array.

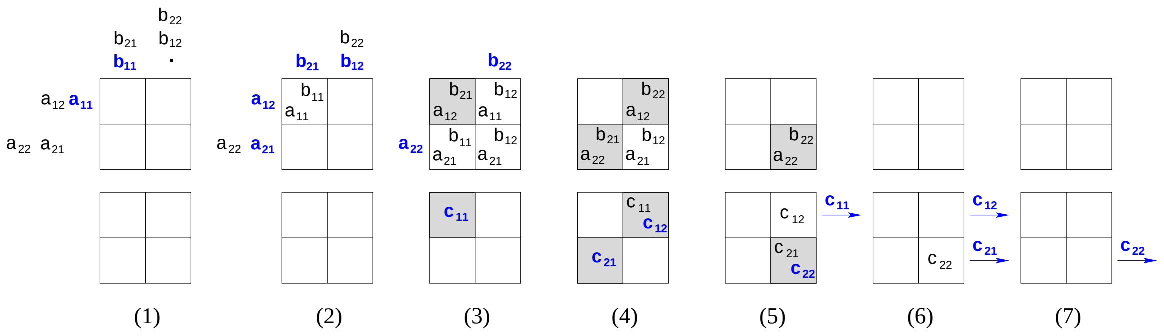


Figure 5-1: Seven time steps of a systolic matrix multiplication $C = A \cdot B$ for 2×2 matrices. Each box represents a compute processor. Values entering, leaving, or being generated in the array are shown in bold face. Shaded boxes mark the completion of an inner product. The data flow of the operands and products is split into the top and bottom rows. The pseudo code executed by each processor in the systolic array is shown in Figure 5-2.

For $N > R$, the compute processor in row r and column s computes the product elements c_{ij} for all $i \bmod R = r$ and $j \bmod R = s$. To supply the compute processors with the proper data streams, one uses R memory processors to store the rows of A and R additional memory processors to store the columns of B. Thus, for the matrix multiplication, one uses $P = R^2$ compute processors and $M = 2R$ memory processors. Figure 5-3 illustrates the data flow of a decoupled systolic matrix multiplication for $N = 4$ and $R = 2$. Note how the memory processors on the periphery determine the schedule of the computations by streaming four combinations

```
$x \leftarrow 0$

**if** $i \neq R$ **and** $j \neq R$
  **for** $n$: 1 .. $N$
    $x \leftarrow x +$ Net($north$) $\times$ Net($west$) **route** Net($north$) $\rightarrow$ Net($south$),
                                              Net($west$) $\rightarrow$ Net($east$)
  **for** $n$: 1 .. $j - 1$
    nop **route** Net($west$) $\rightarrow$ Net($east$)
  Net($east$) $\leftarrow x$

**else if** $i \neq R$ **and** $j = R$
  **for** $n$: 1 .. $N$
    $x \leftarrow x +$ Net($north$) $\times$ Net($west$) **route** Net($north$) $\rightarrow$ Net($south$)
  **for** $n$: 1 .. $j - 1$
    nop **route** Net($west$) $\rightarrow$ Net($east$)
  Net($east$) $\leftarrow x$

**else if** $i = R$ **and** $j \neq R$
  **for** $n$: 1 .. $N$
    $x \leftarrow x +$ Net($north$) $\times$ Net($west$) **route** Net($west$) $\rightarrow$ Net($east$)
  **for** $n$: 1 .. $j - 1$
    nop **route** Net($west$) $\rightarrow$ Net($east$)
  Net($east$) $\leftarrow x$

**else if** $i = R$ **and** $j = R$
  **for** $n$: 1 .. $N$
    $x \leftarrow x +$ Net($north$) $\times$ Net($west$)
  **for** $n$: 1 .. $j - 1$
    nop, **route** Net($west$) $\rightarrow$ Net($east$)
  Net($east$) $\leftarrow x$
```

Figure 5-2: Pseudo code for compute processor p_{ij} executing systolic matrix multiplication. One should note that this algorithm requires only a bounded amount of storage to hold the values x, n, i, j, R, and N.

of rows of A and columns of B into the compute processors. First, C_{11} is computed by streaming $\{A(1,:),\ A(2,:)\}$ and $\{B(:,1),\ B(:,2)\}$ through the array. Second, one streams $\{A(1,:),\ A(2,:)\}$ against $\{B(:,3),\ B(:,4)\}$, third, $\{A(3,:),\ A(4,:)\}$ against $\{B(:,1),\ B(:,2)\}$, and finally $\{A(3,:),\ A(4,:)\}$ against $\{B(:,3),\ B(:,4)\}$. As a result, one computes C_{11}, C_{12}, C_{21}, and C_{22} in that order.

If product matrix C cannot be streamed into a neighboring array of consuming compute processors or off the chip altogether, but shall be stored in memory processors, one may have to invest another R memory processors for a total of $M = 3R$. In any case, $P = \Theta(R^2)$ and $M = \Theta(R)$, and hence $M = o(P)$. Thus, this decoupled systolic matrix multiplication is decoupling efficient.

Note that one could use a similar organization to compute a matrix-vector product Ax, where A is an $N \times N$ matrix and x an $N \times 1$ vector. However, using only one column of $R \times 1$ compute processors requires $M = R + 1$ memory processors. Since $M \neq o(P)$, this organization is not decoupling efficient. However, there exists a different design that is decoupling efficient by storing matrix A and vector x on one memory processor and by distributing the inner products across a linear array of compute processors. Such a distributed inner-product is the key to a decoupling efficient convolution (cf. Chapter 9).

5.3 Efficiency Analysis

The number of multiply-and-add operations in the multiplication of two $N \times N$ matrices is $C(N) = N^3$. On a network of size R with $P = R^2$ compute processors and $M = 2R$ memory processors, one pipelines the computation of $(N/R)^2$ systolic matrix multiplications of size $R \times N$ times $N \times R$. Since this pipelining produces optimal processor utilization, and the startup and drain phases combined take $3R$ time steps (cf. Figure 5-3), the total number of time steps required by this computation is

$$T_{mm}(N,R) \quad = \quad (N/R)^3 R + 3R.$$

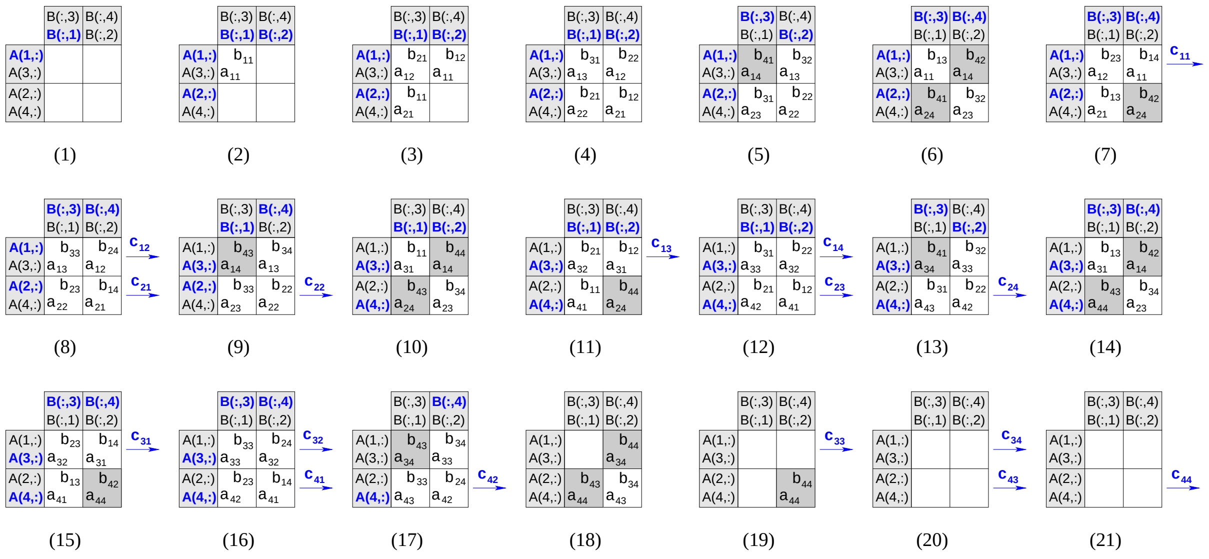


Figure 5-3: Data flow of a compute-efficient matrix multiplication $C = A \cdot B$ for 4×4 matrices on 2×2 compute processors. Shaded boxes on the periphery mark memory processors, and indicate the completion of an inner-product otherwise.

According to Equation 4.1, the floating-point efficiency of the matrix multiplication is therefore

$$E_{mm}(N, R) = \frac{N^3}{((N/R)^3 R + 3R) \cdot (R^2 + 2R)}.$$

Using $\sigma = N/R$ instead of parameter N, one obtains

$$E_{mm}(\sigma, R) = \frac{\sigma^3}{\sigma^3 + 3} \cdot \frac{R}{R+2} \tag{5.3}$$

for the efficiency. Consider each of the two product terms independently. The term $\sigma^3/(\sigma^3 + 3)$ represents the efficiency of the compute processors, and approaches 1 for large values of σ. On the other hand, term $R/(R + 2)$ represents the maximum efficiency expected when using a network of size R. This second term also approaches 1 for large network sizes R. If one assumes a constant value $\sigma \gg 1$, one finds that the efficiency of the matrix multiplication increases with an increase the network size, and approaches the optimal floating-point efficiency of 100 % asymptotically. Also note that for a fixed σ, the stream matrix multiplication requires $T(N) = (\sigma^2 + 3/\sigma)N = \Theta(N)$ time steps on a network with $(N/\sigma)^2$ compute processors.

In practice, the network size R is subject of a delicate trade-off. To maximize efficiency, one wants to maximize both terms in Equation 5.3. Thus, given a problem size N, to increase the first term, one wants to increase $\sigma = N/R$ and, hence, decrease R. On the other hand, to maximize the second term, one wants to increase R. To determine a good value R for implementing a DSA, one may consider some absolute numbers. For example, if $N = R$, that is $\sigma = 1$, one has a systolic matrix multiplication with

$$E_{mm}(\sigma = 1, R) = \frac{1}{4} \cdot \frac{R}{R+2}.$$

Thus, the maximum efficiency is just 25 % even for an infinitely large network. On the other hand, for a relatively small value $\sigma = 8$, one has

$$E_{mm}(\sigma = 8, R) = 0.99 \cdot \frac{R}{R+2}.$$

Hence, for a network size of $R = 16$, a compute-efficient matrix multiplication of problem size $N = 8 \cdot 16 = 128$ achieves almost 90 % efficiency. Larger problem sizes and larger networks operate above 90 % efficiency.

Chapter 6

Triangular Solver

A triangular solver computes the solution x of a linear system of equations $Ax = b$ assuming that matrix A is triangular. Consider an example with a 4×4 lower-triangular matrix A.

$$\begin{pmatrix} a_{11} & 0 & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 \\ a_{31} & a_{32} & a_{33} & 0 \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix}$$

Finding solution x is a straightforward computation known as ***forward substitution***:

$$\begin{aligned} x_1 &= \frac{b_1}{a_{11}} \\ x_i &= \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j \right) \qquad \text{for} \quad i = 2, 3, \ldots, N. \end{aligned}$$

Triangular solvers serve as building blocks of other stream algorithms including LU factorization. Of particular interest is the lower-triangular version that finds an $N \times N$ matrix X as the solution of $AX = B$, where B is an $N \times N$ matrix representing N right-hand sides.

6.1 Partitioning

One should partition the lower-triangular system of linear equations with multiple right-hand sides recursively according to Equation 6.1[1]. Matrices A_{11} and A_{22} are lower triangular.

$$\begin{pmatrix} A_{11} & 0 \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{pmatrix} = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \tag{6.1}$$

The partitioned triangular form leads to a series of smaller problems for the lower-triangular solver:

$$\begin{aligned} A_{11}X_{11} &= B_{11} && (6.2) \\ A_{11}X_{12} &= B_{12} && (6.3) \\ B'_{21} &= B_{21} - A_{21}X_{11} && (6.4) \\ B'_{22} &= B_{22} - A_{21}X_{12} && (6.5) \\ A_{22}X_{21} &= B'_{21} && (6.6) \\ A_{22}X_{22} &= B'_{22} && (6.7) \end{aligned}$$

First, one computes the solution of the lower-triangular systems in Equations 6.2 and 6.3, yielding X_{11} and X_{12}. These solutions are used subsequently to update matrices B_{21} and B_{22} in Equations 6.4 and 6.5, producing B'_{21} and B'_{22}. One may compute the matrix subtraction in Equations 6.4 and 6.5 on the compute processors of the array. However, the associated data movement can be saved by executing the subtraction on the memory processors. This alternative is simpler to program as well. Matrices B'_{21} and B'_{22} are the right-hand sides of the lower-triangular systems in Equations 6.6 and 6.7. Solving these systems yields X_{21} and X_{22}. Thus, Equations 6.2–6.7 define a recursive algorithm for solving the lower-triangular system of Equation 6.1. The recursion reduces the problem of solving a lower-triangular system of linear equations into four smaller lower-triangular systems of linear equations, plus

[1] Irony and Toledo proposed this partitioning of the a triangular solver [17].

two matrix multiplications that have been discussed in Chapter 5 already.

6.2 Decoupling

To arrive at a decoupled design, one begins by observing that the computations for the individual right-hand sides of the linear system $AX = B$ are independent. Consider the following system for $N = 3$ and two right-hand sides.

$$\begin{pmatrix} a_{11} & 0 & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ x_{31} & x_{32} \end{pmatrix} = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{pmatrix}.$$

The computation of column j of X depends on the elements of A and the elements of column j of B only, which means that computations of columns of X may be performed independently.

Figure 6-1 depicts the data flow of the systolic algorithm for a lower-triangular solver, while Figure 6-2 shows pseudocode executed by each compute processor in the systolic array. Rows of A are streamed from the left to the right and columns of B from the top to the bottom of the compute array. Columns of X stream from the bottom of the array. The processor p_{ij} in row i and column j of the compute array is responsible for computing element x_{ij}. Note that due to the independence of columns in this computation one may permute the columns of B arbitrarily, provided the staggered data movement is preserved. One can also use the systolic design of Figure 6-1 for an upper-triangular solver by reversing the order in which the rows of A are stored on the memory processors, and by reversing the order in which the elements of the columns of B are fed into the compute processors.

The systolic algorithm is illustrated by describing the computation of element $x_{31} = (b_{31} - a_{31}x_{11} - a_{32}x_{21})/a_{33}$, beginning with time step 4 in Figure 6-1. Processor p_{31} receives element x_{11} from p_{21} above and a_{31} from the left, and computes the intermediate result $s = a_{31} \cdot x_{11}$. At time step 5, processor p_{31} receives element x_{21} from above and a_{32} from the left. Executing a multiply-and-add operation, p_{31} computes

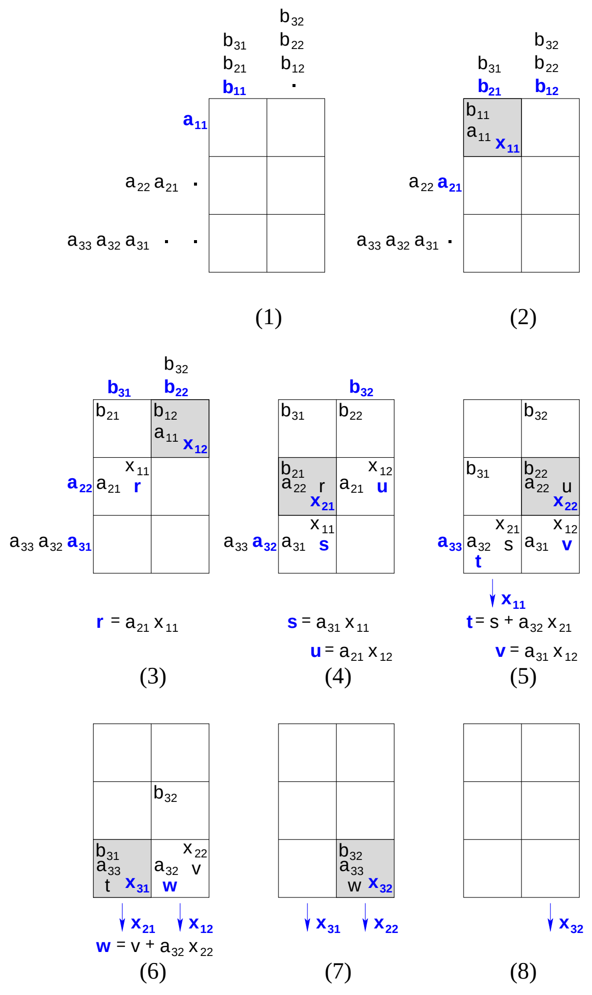


Figure 6-1: Systolic lower-triangular solver for $N = 3$ and two right-hand sides. The pseudocode executed by each processor in the systolic array is shown in Figure 6-2.

intermediate result $t = s + a_{32} \cdot x_{21}$. At time step 6, processor p_{31} receives a_{33} from the left and b_{31} from p_{21} above, and computes $x_{31} = (b_{31} - t)/a_{33}$. During the next time step 7, element x_{31} is available at the bottom of the array.

When reducing a problem of size $N \times N$ recursively until the subproblems fit into an $R \times R$ array of compute processors, one needs $3R$ memory processors on the periphery of the compute array to buffer matrices A, B, and X. Figure 6-3 shows the computation of X_{11} and X_{21} by means of Equations 6.2, 6.4, and 6.6. As implied by this figure, R memory processors are used to store the rows of A, and R memory processors store the columns of B and X, respectively. Thus, for a decoupled systolic lower-triangular solver, one requires $P = R^2$ compute processors and $M = 3R$

```
$x \leftarrow 0$

**if** $j \neq R$
  **for** $n$: $1 \,..\, j-1$
    $x \leftarrow x + \text{Net}(north) \times \text{Net}(west)$ **route** $\text{Net}(north) \rightarrow \text{Net}(south)$,
                                                        $\text{Net}(west) \rightarrow \text{Net}(east)$
  $\text{Net}(south) \leftarrow (\text{Net}(north) - x)/\text{Net}(west)$
  **for** $n$: $j+1 \,..\, R$
    nop **route** $\text{Net}(north) \rightarrow \text{Net}(south)$
**else**
  **for** $n$: $1 \,..\, j-1$
    $x \leftarrow x + \text{Net}(north) \times \text{Net}(west)$ **route** $\text{Net}(north) \rightarrow \text{Net}(south)$
  $\text{Net}(south) \leftarrow (\text{Net}(north) - x)/\text{Net}(west)$
```

Figure 6-2: Pseudo code for compute processor p_{ij} executing systolic lower triangular solver. One should note that each processor requires a bounded amount of storage to hold the values x, n, j, R.

memory processors, meeting the decoupling-efficiency condition $M = o(P)$.

Unlike the matrix multiplication, the factorization of the triangular solver does not produce identical subproblems. Therefore, one is faced with the additional challenge of finding an efficient composition of these subproblems. Although the subproblems can be pipelined, one cannot avoid idle cycles due to data dependencies and the heterogeneity of the computations in Equations 6.2–6.7. However, this loss of cycles can be minimized by grouping independent computations of the same type, and pipelining those before switching to another group. For example, one can group and pipeline the computations of Equations 6.2 and 6.3, then Equations 6.4 and 6.5, and finally Equations 6.6 and 6.7. If one unfolds the recursion all the way to the base case of $R \times R$ subproblems, the best schedule is equivalent to a block-iterative ordering of the subproblems.

6.3 Efficiency Analysis

The efficiency of the lower-triangular solver is computed according to Equation 4.1. The number of floating-point multiply-and-add operations is $C(N) = N^3/2$, assuming

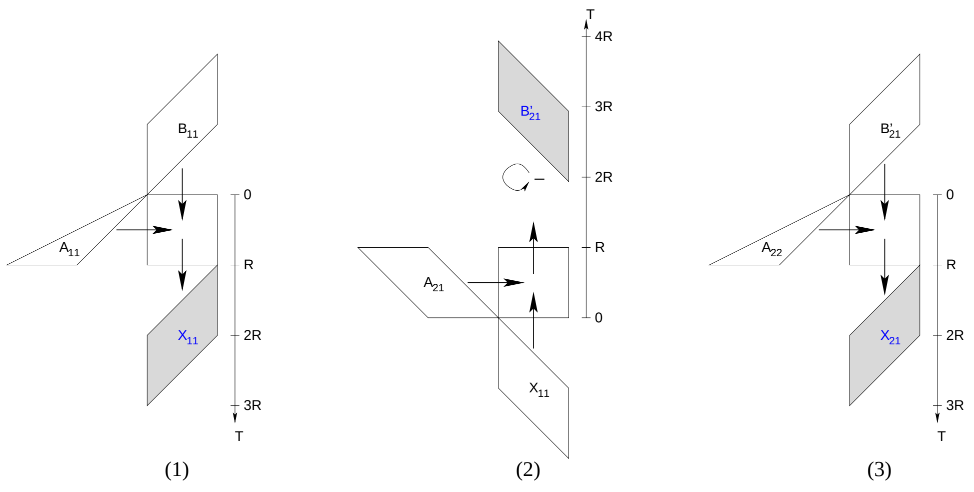


Figure 6-3: Phases of a decoupled systolic lower-triangular solver on an $R \times R$ array of compute processors. In phase 1, Equation 6.2 is solved for X_{11}. In phase 2, B_{21} is updated according to Equation 6.4. While the matrix multiplication is executed on the compute processors, the matrix subtraction is performed on the memory processors as they receive each individual result of $A_{21}X_{11}$. Finally, in phase 3, Equation 6.6 is solved for X_{21}. The shapes of the matrix areas indicate how the rows and columns enter the compute array in a staggered fashion.

that a division is no more costly than a multiply-and-add operation.

To calculate the number of time steps required by the stream-structured triangular solver, one uses a block-iterative schedule to group and pipeline independent subproblems. The partitioning is applied until A, X, and B are $\sigma \times \sigma$ block matrices where each block is an $R \times R$ matrix. As an example of the block-iterative schedule, consider the case when $\sigma = 4$ represented by Equation 6.8.

$$\begin{pmatrix} A_{11} & 0 & 0 & 0 \\ A_{21} & A_{22} & 0 & 0 \\ A_{31} & A_{32} & A_{33} & 0 \\ A_{41} & A_{42} & A_{43} & A_{44} \end{pmatrix} \begin{pmatrix} X_{11} & X_{12} & X_{13} & X_{14} \\ X_{21} & X_{22} & X_{23} & X_{24} \\ X_{31} & X_{32} & X_{33} & X_{34} \\ X_{41} & X_{42} & X_{43} & X_{44} \end{pmatrix} = \begin{pmatrix} B_{11} & B_{12} & B_{13} & B_{14} \\ B_{21} & B_{22} & B_{23} & B_{24} \\ B_{31} & B_{32} & B_{33} & B_{34} \\ B_{41} & B_{42} & B_{43} & B_{44} \end{pmatrix} \tag{6.8}$$

Consider the computation of row block i of X, that is X_{ij}. This row block is computed after blocks X_{kj} where $k < i$ and after updates to B. These updates are made by computing $B'_{kj} = B_{kj} - \sum_{l=0}^{k} A_{kl}X_{lj}$ for all k and j where $k < i$ and $1 \le j \le 4$. To compute row block i of X, one then solves $A_{ii}X_{ij} = B_{ij}$ for all j where $1 \le j \le 4$.

Having completed this step, one must further update all B' values by computing $B''_{kj} = B'_{kj} - A_{ki}X_{ij}$ for all k and j where $i < k, j \leq 4$.

In general, one wants to determine the number of time steps required to compute a row block of X for an arbitrary value of σ. Computing the row block i of X requires a single triangular solver for each $R \times R$ submatrix in the row block (as in Equations 6.2–6.3), and there are σ such blocks in each row. Each triangular solver is independent, thus all the solvers for one row can be pipelined. Because the inputs into the array are staggered, it takes R time steps to fill the pipeline and R more to drain it. When, executing in steady-state, each triangular solver takes R time steps to execute. Therefore, each block of rows in X requires $\sigma R + 2R$ time steps for the triangular solvers. Once the values of X are found, one must update the rows of B as in Equations 6.4–6.5. There are $\sigma - i$ row blocks of B that need to be updated, and each block requires σ multiplications of $R \times R$ matrices, all of which are independent and can be pipelined. R time steps are needed to start the matrix multiplication pipeline, $2R$ to drain it, and a total of $\sigma(\sigma - i)$ matrix multiplications will be performed. Thus, updating the remaining blocks of the matrix requires $\sigma(\sigma-i)R+3R$ time steps. One may overlap the computation of the triangular solvers and the matrix multiplications by R time steps each.

One computes the total number of time steps required by the stream-structured lower-triangular solver by summing the time steps required to compute each row block of X. There are σ row blocks. Thus, the total execution time is the sum of all the time steps needed for the solvers for each row block plus the sum of all time steps needed for matrix multiplications for each row block minus the sum of all time steps of overlap between the solvers and the matrix multiplications:

$$\begin{aligned} T_{lts}(\sigma, R) &= \sum_{i=1}^{\sigma}(\sigma R + 2R) + \sum_{i=1}^{\sigma-1}(\sigma(\sigma - i)R + 3R) - \sum_{i=1}^{\sigma-1} R \\ &= \frac{R}{2}(\sigma^3 + \sigma^2 + 6\sigma - 2). \end{aligned}$$

For a fixed σ, the total number of time steps is $T(N) = \Theta(N)$ when using $(N/\sigma)^2$ compute processors.

According to Equation 4.1, the floating-point efficiency of the compute-efficient lower-triangular solver is then

$$E_{lts}(\sigma, R) \quad = \quad \frac{\sigma^3}{\sigma^3 + \sigma^2 + 6\sigma - 2} \cdot \frac{R}{R+3}. \tag{6.9}$$

Analogous to Equation 5.3 for the matrix multiplication, the efficiency is the product of two terms, one depending indirectly on the problem size N via σ, and the second depending on the network size R. Again, the first term represents the efficiency of the compute processors, while the second term represents the maximum efficiency given the network size. When $\sigma = 1$, the problem reduces to a single systolic lower-triangular solver, and one obtains an efficiency of

$$E_{lts}(\sigma = 1, R) = \frac{1}{6} \cdot \frac{R}{R+3}.$$

The efficiency increases when one increases R and σ, such that the floating-point efficiency approaches the optimal value of 100 %. Since the solver requires memory processors along three sides of the array of compute processors, the second term requires a slightly larger network size to achieve high efficiency. For example, for a very large σ, $E_{lts}(R) \approx R/(R+3)$, and one achieves more than 90 % efficiency for $R > 27$.

Chapter 7

LU Factorization

The LU factorization transforms an $N \times N$ matrix A into two $N \times N$ matrices L and U, such that L is lower-triangular, U is upper-triangular, and $A = LU$. Furthermore, this treatment of the LU factorization requires that for all diagonal elements of $L = (l_{ij})$, $l_{ii} = 1$, where $1 \leq i \leq N$.

7.1 Partitioning

The LU factorization is partitioned according to Equation 7.1[1]. Matrices L_{11} and L_{22} are lower triangular, while matrices U_{11} and U_{22} are upper triangular.

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{pmatrix} \tag{7.1}$$

This partitioning results in a series of smaller problems:

$$A_{11} = L_{11}U_{11} \tag{7.2}$$

$$A_{12} = L_{11}U_{12} \tag{7.3}$$

$$A_{21} = L_{21}U_{11} \tag{7.4}$$

$$A'_{22} = A_{22} - L_{21}U_{12} \tag{7.5}$$

[1]Irony and Toledo proposed this partitioning of the LU factorization [17].

$$A'_{22} = L_{22}U_{22} \tag{7.6}$$

First, one computes L_{11} and U_{11} by factoring A_{11} according to Equation 7.2. These results are used to solve Equations 7.3 and 7.4 for U_{12} and L_{21} respectively. Then, A_{22} is updated according to Equation 7.5, which produces A'_{22}. As with the triangular solver, the matrix subtraction required by this step can be performed on the memory tiles. Finally, A'_{22} is used to compute L_{22} and U_{22}. The recursive formulation due to Equations 7.2–7.6 reduces the problem of an LU factorization into two smaller LU factorizations, a matrix multiplication, a lower-triangular solver, and an upper-triangular solver of the form $XU = B$. The stream-structured version of this upper-triangular solver is similar to that of the lower-triangular solver.

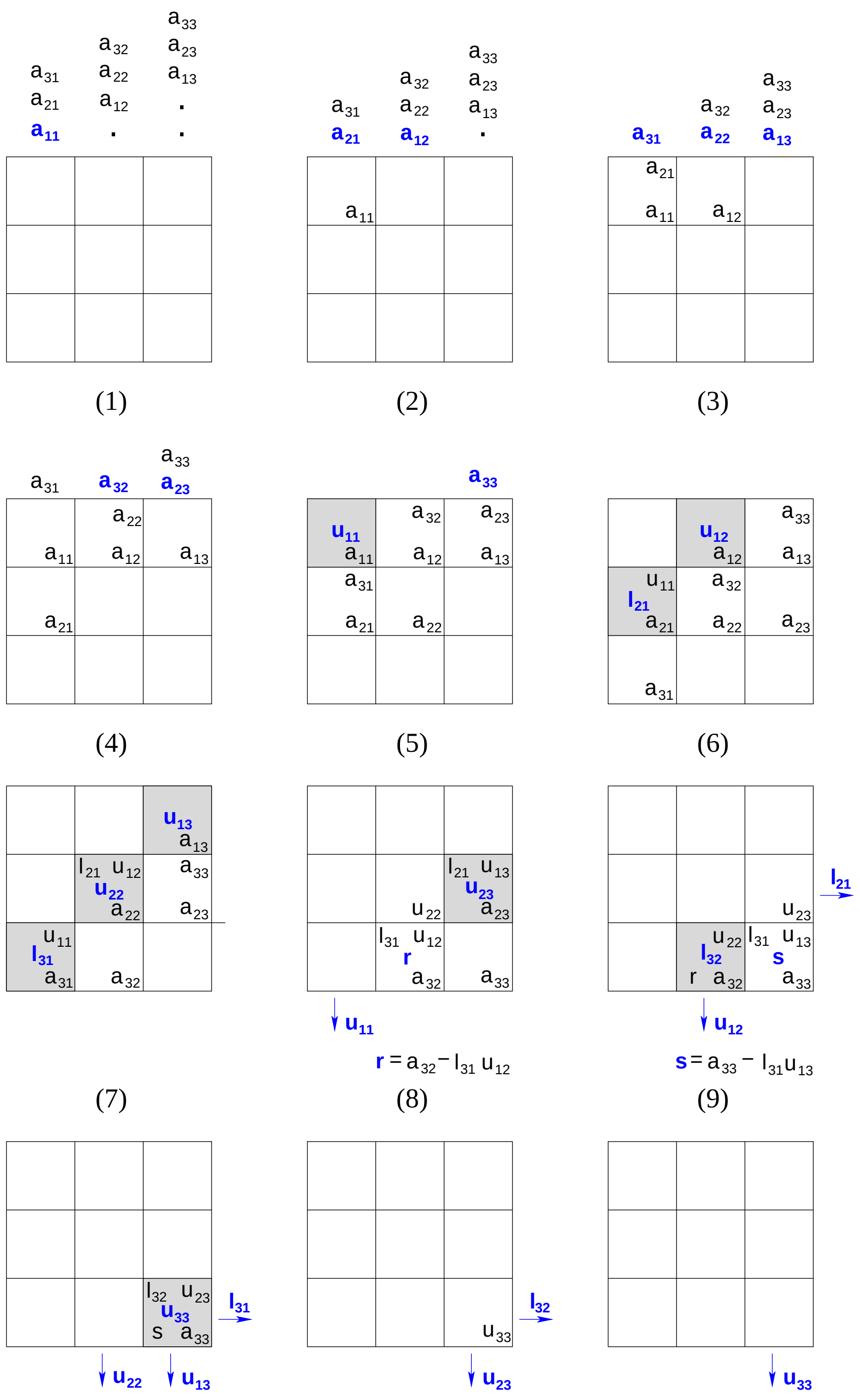
7.2 Decoupling

The derivation of a decoupled design begins with a systolic LU factorization. As an example, consider the LU factorization for $N = 3$.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix}$$

Figure 7-1 shows the progress of the systolic LU factorization, while Figure 7-2 shows the pseudocode executed by each compute processor. Columns of matrix A enter the compute array at the top, and fold over towards the bottom. The columns of upper-triangular matrix U leave the array at the bottom and the rows of lower-triangular matrix L exit on the right. The compute processor p_{ij} in row i and column j of the compute array computes either u_{ij} if $i < j$, or l_{ij} otherwise. Since $l_{ii} = 1$, the diagonal elements of L are neither computed nor stored explicitly.

The data flow pattern of the LU factorization is straightforward. Elements of L stream from left to right, and elements of U stream from top to bottom. When a pair of elements enters processor p_{ij}, it computes an intermediate value of either l_{ij} or u_{ij}.



(1) (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (12)

Figure 7-1: Systolic LU factorization for $N = 3$. Figure 7-2 shows the pseudocode executed by each compute processor performing a systolic LU factorization.

```
$x \leftarrow$ Net($north$)

if $i \leq j$ // computing U
  for $n$: 1 .. $i-1$
    $x \leftarrow x -$ Net($west$) $\times$ Net($north$) route Net($west$) $\rightarrow$ Net($east$),
                                                  Net($north$) $\rightarrow$ Net($south$)
  Net($south$) $\leftarrow x$

if $i > j$ // computing L
  for $n$: 1 .. $j-1$
    $x \leftarrow x -$ Net($west$) $\times$ Net($north$) route Net($west$) $\rightarrow$ Net($east$),
                                                  Net($north$) $\rightarrow$ Net($south$)
  Net($east$) $\leftarrow x/$Net($north$)
```

Figure 7-2: Pseudo code for compute processor p_{ij} executing systolic LU factorization. Note that the storage of values i, j, x, and n requires a bounded amount of storage only.

As a concrete example, consider the computation of element $u_{22} = a_{22} - l_{21} \cdot u_{12}$, where $u_{12} = a_{12}$, $u_{11} = a_{11}$, and $l_{21} = a_{21}/u_{11}$. Processor p_{22} at the center of the array will produce value u_{22}. At time step 6 of Figure 7-1, processor p_{21} receives u_{11} from above and uses it to compute element l_{21}. This value is sent to the right and becomes available on processor p_{22} at time step 7. Simultaneously, processor p_{12} sends $u_{12} = a_{12}$ downwards towards processor p_{22}. At time step 7, processor p_{22} receives elements u_{12} from p_{12} above and l_{21} from p_{21} on the left. With element a_{22} already resident since time step 5, processor p_{22} computes $u_{22} = a_{22} - l_{21} \cdot u_{12}$. Value u_{22} remains on processor p_{22} during time step 7, while value u_{12} is sent towards neighbor p_{32}. Then, during time step 8, u_{12} is sent to neighbor p_{32}. At time step 9 processor p_{32} uses u_{22} to compute $l_{32} = (a_{32} - l_{31} \cdot u_{12})/u_{22}$. In time step 10, element u_{22} leaves the array at the bottom of processor p_{32}.

Analogous to the triangular solver, one reduces a problem of size $N \times N$ recursively until the subproblems fit into an $R \times R$ array of compute processors. Figure 7-3 illustrates the data movement of the matrices when computing five systolic subproblems according to Equations 7.2–7.6. The algorithm needs R memory processors to buffer the columns of A, another R for the rows of L, and an additional R for the columns of U. Thus, the decoupled systolic LU factorization requires $P = R^2$ compute processors and $M = 3R$ memory processors. One observes that $M = o(P)$, and the structure of the LU factorization is decoupling efficient.

Similar to the lower-triangular solver, the partitioning due to Equations 7.2–7.6 produces a set of heterogeneous subproblems. To obtain the most efficient composition of the subproblems, one may group, pipeline, and overlap independent subproblems. Unfolding the recursion all the way to subproblems of size R permits a block-iterative schedule with efficient pipelining. Analogous to the standard threefold loop of the LU factorization, one can solve Equation 7.2 for each pivot block, pipeline the solvers of Equations 7.3 and 7.4 for the pivot row and column, and update the lower right matrix by pipelining the matrix multiplications of Equation 7.5. The computation of Equation 7.6 corresponds to the factorization of the next pivot block.

7.3 Efficiency Analysis

The number of multiply-and-add operations of an $N \times N$ LU factorization is $C(N) \approx N^3/3$. This approximation neglects an asymptotically insignificant linear term, if we count divisions as multiply-and-add operations.

One may calculate the number of time steps required for our stream-structured LU factorization when using a block-iterative schedule and pipelining independent subproblems. The computation is partitioned until A, L, and U are $\sigma \times \sigma$ block matrices where each block is itself an $R \times R$ matrix. As an example of this schedule, consider the case when $\sigma = 4$ represented by Equation 7.7.

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 & 0 & 0 \\ L_{21} & L_{22} & 0 & 0 \\ L_{31} & L_{32} & L_{33} & 0 \\ L_{41} & L_{42} & L_{43} & L_{44} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} & U_{14} \\ 0 & U_{22} & U_{23} & U_{24} \\ 0 & 0 & U_{33} & U_{34} \\ 0 & 0 & 0 & U_{44} \end{pmatrix} \tag{7.7}$$

Consider the computation of L_{22} and U_{22}. Following the partitioning, this calculation

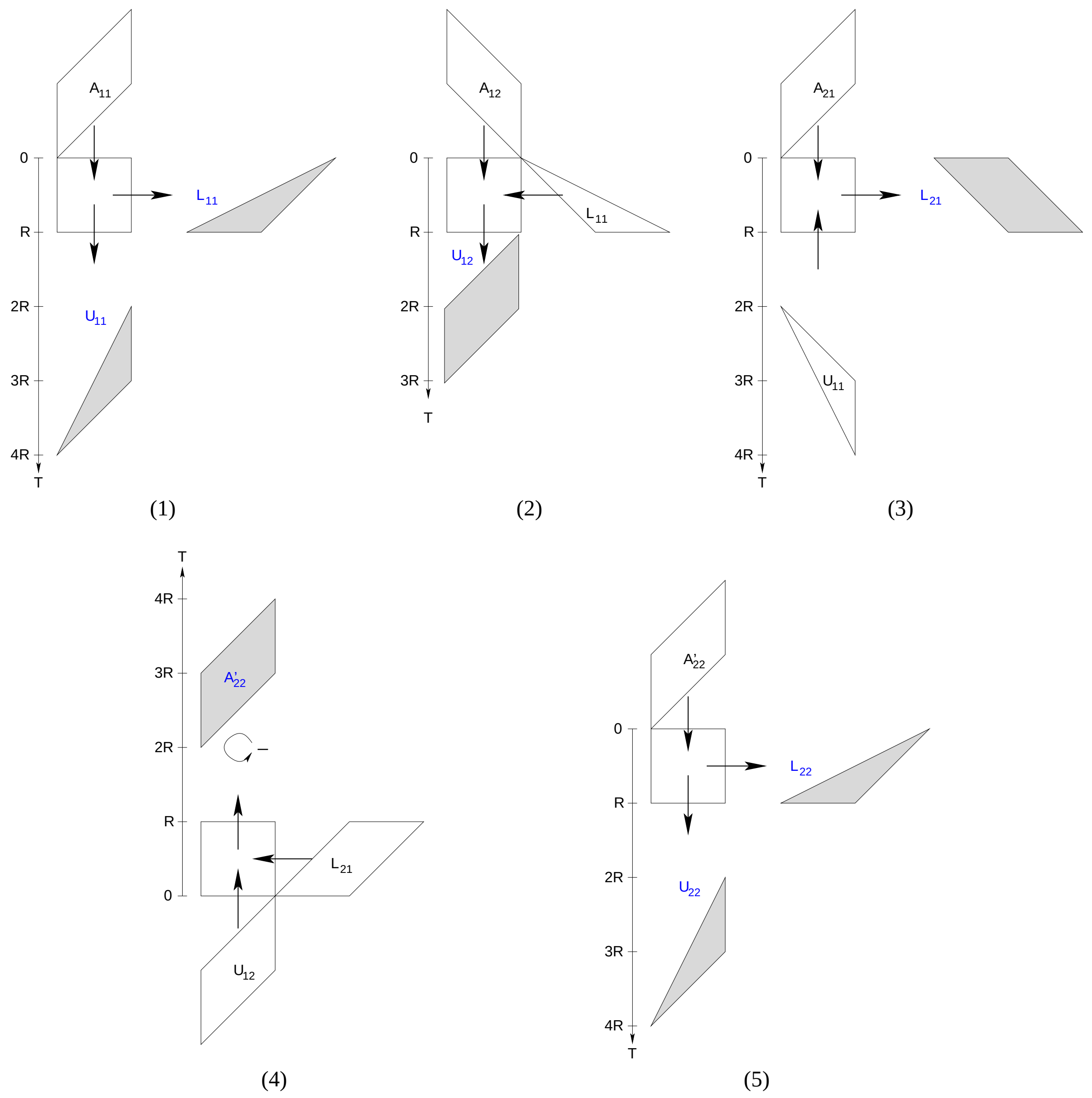


Figure 7-3: Phases of stream-structured LU factorization on an $R \times R$ array of compute processors, with $N = 2R$. Phase 1 factors A_{11} into L_{11} and U_{11} according to Equation 7.2. Phase 2 solves Equation 7.3 for U_{12}. Phase 3 solves Equation 7.4 for L_{21}. Phase 4 computes A'_{22} according to Equation 7.5, performing the matrix subtraction on the memory tiles. Finally, in phase 5 A'_{22} is factored according to Equation 7.6. The shapes of the matrices indicate that the rows and columns enter the compute array in a staggered fashion.

is performed after the first column block of L and the first row block of U have been computed and A has been updated by computing $A'_{ij} = A_{ij} - L_{i1}U_{1j}$ where $2 \leq i, j \leq 4$. Thus, one can compute the factorization $A'_{22} = L_{22}U_{22}$. Next one determines the second column block of L by solving the lower-triangular systems $A'_{i2} = L_{i2}U_{22}$ and the second row block of U is determined by solving the upper-triangular systems $A'_{2j} = L_{22}U_{2j}$ where $3 \leq i, j \leq 4$. Finally, the remaining part of the matrix is updated by computing $A''_{ij} = A'_{ij} - L_{i2}U_{2j}$ where $3 \leq i, j \leq 4$.

It is now possible to determine the number of time steps required to compute diagonal blocks of L and U for an arbitrary value of σ. Computing diagonal blocks L_{ii} and U_{ii} requires a single LU factorization, which takes $4R$ time steps. Once these values are computed, they can be used to calculate the $\sigma - i$ blocks of row block i of U and the $\sigma - i$ blocks of column block i of L as in Equations 7.3 and 7.4, respectively. Computing these superdiagonal values of U requires a lower-triangular solver for each block, and the computation of each block can be pipelined. There are R time steps required to fill the pipeline, and R time steps to drain it while $\sigma - i$ problems will be pipelined. Thus, a total of $(\sigma - i)R + 2R$ time steps are required for computing the superdiagonal elements in row block i of U. Computing the sub diagonal elements of L requires an upper-triangular solver for each block in the column, and again the computation can be pipelined. In this case, it takes an additional R time steps to start the pipeline because the matrices A and U are not adjacent (cf. Figure 7-3(c)). Thus, a total of $(\sigma - i)R + 3R$ time steps are required for computing the subdiagonal blocks in column i of L. Having computed these values of L and U, one must update the A matrix as in Equation 7.5. The $(\sigma - i)R \times (\sigma - i)R$ submatrix in the lower right corner of A is updated by performing matrix multiplications and subtractions as in Equation 7.6. All matrix multiplications are independent, so one may pipeline these problems. R cycles are required to fill the pipeline and an additional $2R$ are needed to drain it. Due to the size of the submatrix being updated, the algorithm performs a total of $(\sigma - i)^2$ matrix multiplications. Thus, the updates of matrix A require $(\sigma - i)^2R + 3R$ time steps. One may overlap the computation of each phase of the partitioning to increase the efficiency. There are R cycles of overlap between the lower-triangular solver and the LU factorization. There are $2R$ cycles of overlap between the upper- and lower-triangular solvers. There are R cycles of overlap between the matrix multiplication and the and the upper-triangular solver. Finally, there are $2R$ cycles of overlap between the subsequent LU factorization and the matrix multiplication.

The total number of time steps required by the stream-structured LU factorization is computed by summing the number of cycles required to compute each (L_{ii}, U_{ii}) pair. Thus the total number of time steps is the sum of the number of time steps spent performing LU factorizations plus the sum of the number of time steps dedicated to lower-triangular solvers plus the total number of time steps spent computing upper-triangular solvers, plus the total number of time steps spent performing matrix multiplications minus the time steps where computations can be overlapped:

$$\begin{aligned} T_{lu}(\sigma, R) &\approx \sum_{k=1}^{\sigma} 4R \\ &\quad + \sum_{k=1}^{\sigma-1} ((\sigma-k)R + 2R) + \sum_{k=1}^{\sigma-1} ((\sigma-k)R + 3R) + \sum_{k=1}^{\sigma-1} \left((\sigma-k)^2 R + 3R\right) \\ &\quad - \sum_{k=1}^{\sigma-1} 6R \\ &= R\left(\frac{1}{3}\sigma^3 + \frac{1}{2}\sigma^2 + \frac{31}{6}\sigma - 2\right). \end{aligned}$$

One observes that for a fixed σ, the total number of time steps is $T(N) = \Theta(N)$ when using $(N/\sigma)^2$ compute processors.

Using a network of size R, $P = R^2$ compute processors, and $M = 3R$ memory processors, the floating-point efficiency of the LU factorization according to Equation 4.1 is

$$E_{lu}(\sigma, R) \approx \frac{\sigma^3}{\sigma^3 + \frac{3}{2}\sigma^2 + \frac{31}{2}\sigma - 6} \cdot \frac{R}{R+3}.$$

Analogous to the treatment of E_{mm} and E_{lts} in Equations 5.3 and 6.9, one may split E_{lu} into two terms, one of which depends on $\sigma = N/R$, and one depending on the network size R. For $\sigma = 1$, the problem reduces to a single systolic LU factorization, and one obtains a compute efficiency of

$$E_{lu}(\sigma = 1, R) = \frac{1}{12} \cdot \frac{R}{R+3}.$$

Asymptotically, that is for large R and σ, the compute efficiency of the LU factorization on the DSA approaches the optimal value of 100 %. As for the triangular-solver, when $\sigma \gg 1$, $E_{lu}(R) \approx R/(R+3)$, and one achieves more than 90 % efficiency for $R \geq 27$.

Chapter 8

QR Factorization

QR factorization decomposes an $M \times N$ matrix A with $M \geq N$ into two matrices Q and R, such that Q is an orthogonal $M \times M$ matrix, R is an upper triangular $M \times N$ matrix, and $A = QR$. This treatment assumes that the factorization is based on fast Givens rotations [11]. The application of a fast Givens rotation, expressed as a matrix multiplication, annihilates an element of A. The $M \times M$ matrix $G(i,j)$ is a ***fast Givens transformation***, if it places value zero in element (i,j) of the product $G(i,j)^T \cdot A$. One may restrict $G(i,j) = (g_{\hat{\imath}\hat{\jmath}})$ to the form

$$G(i,j) \quad = \quad \begin{array}{c} \\ \\ \\ j \\ \\ i \\ \\ \\ \end{array} \begin{array}{c} \begin{array}{ccccccc} & & j & & i & & \end{array} \\ \left(\begin{array}{ccccccc} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & \alpha & & \\ & & & \ddots & & & \\ & & \beta & & 1 & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{array} \right) \end{array}, \qquad g_{\hat{\imath}\hat{\jmath}} = \begin{cases} 1\,, & \hat{\imath} = \hat{\jmath} \\ \alpha, & \hat{\imath} = j \wedge \hat{\jmath} = i \\ \beta, & \hat{\imath} = i \wedge \hat{\jmath} = j \\ 0\,, & \text{otherwise.} \end{cases} \tag{8.1}$$

All diagonal elements of $G(i,j)$ have value 1 and all off-diagonal elements have value 0, with the exception of elements g_{ji} and g_{ij}, which have non-zero values $g_{ji} = \alpha$ and $g_{ij} = \beta$. Given matrix $A = (a_{ij})$ and an auxiliary diagonal $M \times M$ matrix $D = (d_i)$,

α and β are determined as $\alpha = -a_{ij}/a_{jj}$, $\beta = -\alpha d_j/d_i$, $\gamma = -\alpha\beta$, $d_i = (1+\gamma)d_i$, and $d_j = (1+\gamma)d_j$. Matrix D allows one to compute α and β without square root operations. One should initialize D such that $d_i = 1$ for $1 \le i \le M$.

One may apply a series of fast Givens transformations to A to create an upper triangular matrix. Since transformation $G(i,j)$ annihilates element a_{ij}, one applies $MN - N^2/2 - N/2$ fast Givens transformations with $1 \le j \le N$ and $j < i \le M$ to triangularize matrix A. Once a particular sequence of fast Givens transformations is chosen, the computation of D and Q must observe this order. This treatment assumes a sequence of fast Givens transformations that generates an upper triangular matrix U by annihilating the elements below the diagonal column-wise from left to right, and within each column from top to bottom. More succinctly, one applies the sequence of fast Givens transformations $G(2,1)$, $G(3,1)$, …, $G(M,1)$ to annihilate all elements below the diagonal in the first column, then proceeds by transforming the second column with $G(3,2)$, $G(4,2)$, …, $G(M,2)$, and so on up to column N:

$$G(M,N)^T \cdots G(N+1,N)^T \cdots G(M,2)^T \cdots G(3,2)^T G(M,1)^T \cdots G(2,1)^T A \;=\; U.$$

Since $(AB)^T = B^T A^T$, this product can be written in compact form as[1]

$$G^T A = U, \quad \text{where} \quad G = \prod_{j=1}^{N} \prod_{i=j+1}^{M} G(i,j).$$

Finally, one should consider the role of diagonal matrix D briefly. By construction of the fast Givens transformation, it is the case that $G^T G = D$. Since D is diagonal, one may split D such that $D = D^{1/2} D^{1/2}$. Then one obtains $D^{-1/2} G^T G D^{-1/2} =$

[1]In expressing products of Givens transformations, the order of operations effects efficiency. The convention used here is that the product notation defines a sequence of multiplications that corresponds to the sequence of indices in the product such that multiplications with larger indices are applied from the right, that is, one interprets products as right-associative and the matrix multiplication as left-associative to determine the sequence uniquely. For example,

$$\prod_{j=1}^{2} \prod_{i=j+1}^{3} G(i,j) = \prod_{j=1}^{2} \left(\prod_{i=j+1}^{3} G(i,j) \right) = (G(2,1) \cdot G(3,1)) \cdot G(3,2).$$

$(GD^{-1/2})^T(GD^{-1/2}) = I$, and $GD^{-1/2}$ is orthogonal. Thus, one may rewrite $G^TA = U$ as $(D^{-1/2}G^T)A = D^{-1/2}U$ with the consequence that $Q = GD^{-1/2}$ is orthogonal and $R = D^{-1/2}U$ is upper triangular.

8.1 Partitioning

One may partition the QR factorization according to Equation 8.2 for $M \geq N^2$. Without loss of generality, the discussion is restricted to the case where $M = 3/2N$, such that matrices A_{ij} and R_{ij} in Equation 8.2 are $N/2 \times N/2$ matrices.

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \\ A_{31} & A_{32} \end{pmatrix} = Q \begin{pmatrix} R_{11} & R_{12} \\ 0 & R_{22} \\ 0 & 0 \end{pmatrix} \tag{8.2}$$

This partitioning allows one to formulate the QR factorization as a series of three subproblems, defined by Equations 8.3–8.5.

$$\begin{pmatrix} A_{11} \\ A_{21} \\ A_{31} \end{pmatrix} = Q_1 \begin{pmatrix} R_{11} \\ 0 \\ 0 \end{pmatrix} \tag{8.3}$$

$$\begin{pmatrix} R_{12} \\ A'_{22} \\ A'_{32} \end{pmatrix} = Q_1^T \begin{pmatrix} A_{12} \\ A_{22} \\ A_{32} \end{pmatrix} \tag{8.4}$$

$$\begin{pmatrix} A'_{22} \\ A'_{32} \end{pmatrix} = \hat{Q}_2 \begin{pmatrix} R_{22} \\ 0 \end{pmatrix} \tag{8.5}$$

Equations 8.3–8.5 enable one to compute the QR factorization by solving three smaller problems. First, one triangularizes columns $1, \ldots, N/2$ of A according to Equation 8.3. As a result, one obtains R_{11} and a sequence of fast Givens transformations associated with matrix Q_1. Next, one updates columns $N/2 + 1, \ldots, N$

[2]Elmroth and Gustavson proposed this partitioning of the QR factorization [9].

of A according to Equation 8.4, which yields R_{12} and the intermediate matrices A'_{22} and A'_{32}. This computation uses the sequence of fast Givens transformations calculated in the previous step, without computing Q_1 explicitly. Then, one triangularizes columns $N/2+1, \ldots, N$ according to Equation 8.5, resulting in R_{22} and the sequence of fast Givens transformations associated with $\hat{Q}_2$. Matrix $\hat{Q}_2$ is an $N \times N$ submatrix of Q_2. Matrices Q, Q_1, and Q_2 are defined in terms of fast Givens transformations by Equations 8.6–8.8:

$$Q_1 = \left(\prod_{j=1}^{N/2} \prod_{i=j+1}^{M} G(i,j) \right) \hat{D}^{-1/2} \tag{8.6}$$

$$Q_2 = \begin{pmatrix} I & 0 & 0 \\ 0 & & \\ 0 & \multicolumn{2}{c}{\hat{Q}_2} \end{pmatrix} = \left(\prod_{j=N/2+1}^{N} \prod_{i=j+1}^{M} G(i,j) \right) \tilde{D}^{-1/2} \tag{8.7}$$

$$Q = Q_1 Q_2 = \left(\prod_{j=1}^{N} \prod_{i=j+1}^{M} G(i,j) \right) \hat{D}^{-1/2} \tilde{D}^{-1/2}, \tag{8.8}$$

where $\hat{D} = (\hat{d}_i)$ and $\tilde{D} = (\tilde{d}_i)$ are diagonal $M \times M$ matrices. Furthermore, $\hat{d}_i = d_i$ for $0 \leq i \leq N/2$ and $\hat{d}_i = 1$ otherwise, and $\tilde{d}_i = d_i$ for $N/2+1 \leq i \leq M$ and $\tilde{d}_i = 1$ otherwise.

One need not compute the intermediate forms Q_1 and Q_2 of matrix Q in order to triangularize matrix A. Instead, the sparse structure of the fast Givens transformation may be utilized to operate with a highly efficient representation. Recall from the definition of $G(i,j)$ in Equation 8.1 that $G(i,j)$ contains only two characteristic values α and β. Thus, it suffices to represent $G(i,j)$ by means of the pair $(\alpha_{ij}, \beta_{ij})$. In addition to being a space efficient representation, it is also advantageous for implementing the only two operations associated with fast Givens rotations, premultiplication and postmultiplication. the ***premultiplication*** of a matrix A with fast Givens transformation $G(i,j)$ occurs when forming the product $G(i,j)^T \cdot A$, while the ***postmultiplication*** of matrix A occurs when forming the product $A \cdot G(i,j)$. Due to the structure of $G(i,j)$, premultiplication effects rows i and j of A only, and postmultiplication changes the values in columns i and j of A only, cf. Figure 8-1.

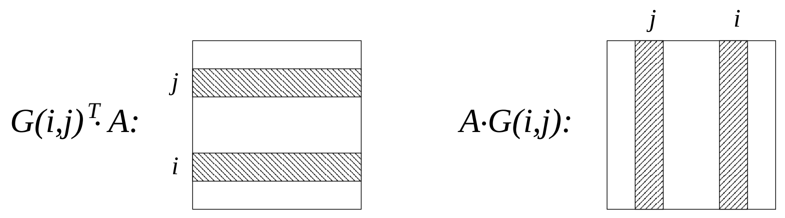


Figure 8-1: The product of premultiplication $G(i,j)^T \cdot A$ differs from A in rows i and j only, while the product of postmultiplication $A \cdot G(i,j)$ differs from A in columns i and j only.

Premultiplication of an $M \times M$ matrix $A = (a_{ij})$ produces elements $a'_{jk} = a_{jk} + \beta_{ij} \cdot a_{ik}$ in row j and $a'_{ik} = a_{ik} + \alpha_{ij} \cdot a_{jk}$ in row i for $1 \leq k \leq M$. All other elements of A remain unchanged by premultiplication. Analogously, postmultiplication results in elements $a'_{kj} = a_{kj} + \beta_{ij} \cdot a_{ki}$ in column j and elements $a'_{ki} = a_{ki} + \alpha_{ij} \cdot a_{kj}$ in column i for $1 \leq k \leq M$. All other elements of A are retained by postmultiplication. One may note that both premultiplications and postmultiplications offer various degrees of parallelism. Specifically, one can exploit the fact that the premultiplications $G(i,j)^T \cdot A$ and $G(k,l)^T \cdot A$ and postmultiplications $A \cdot G(i,j)$ and $A \cdot G(k,l)$ are independent for mutually distinct values of i, j, k and l.

In summary, the partitioning of Equation 8.2 permits one to express the computation of matrix R of the QR factorization as two smaller QR factorizations of Equations 8.3 and 8.5 and a sequence of premultiplications with the fast Givens transformations associated with Q_1^T according to Equation 8.4. If matrix Q is not desired, as may be the case when using the factorization as part of a linear system solver, Q_1 and Q_2 do not have to be computed explicitly. If the Q matrix is desired it can be computed by means of postmultiplications using a partitioning along rows instead of columns. However, in general it will be more efficient to compute with the fast Givens transformations than with an assembled Q matrix.

8.2 Decoupling

The decoupled design for the QR factorization is based on three systolic algorithms, each using an $R \times R$ array of compute processors. The first algorithm performs a ***sys-***

tolic Givens computation by triangularizing an $M \times R$ matrix, where $M \geq R$; cf. Equation 8.3 and, analogously, Equation 8.5. The systolic Givens computation produces a sequence of fast Givens transformations $G(i,j)$, each of which is represented by the pair $(\alpha_{ij}, \beta_{ij})$, and the corresponding values of diagonal matrix D. The second algorithm implements the update operation of Equation 8.4 as a ***systolic premultiplication*** of an $M \times R$ matrix. The third algorithm computes matrix Q by means of ***systolic postmultiplications*** according to Equations 8.6–8.8 on $R \times M$ matrices. Typically, the Q matrix need not be explicitly computed, as many operations with this matrix can be computed more economically by working only with the fast Givens transformations. However, the computation of the Q matrix is still interesting as an example of systolic postmultiplications.

Figures 8-2 and 8-3 illustrate the systolic Givens computation for triangularizing an $M \times R$ matrix A, where $M \geq R$[3]. Figure 8-4 shows the pseudocode executed by the compute processors performing a systolic Givens computation. Columns of matrix A enter the array at the top. In addition, the elements of diagonal matrix D enter the array by interleaving them with the leftmost column of matrix A. The resulting upper-triangular matrix R leaves the array at the bottom. Furthermore, the Givens computation yields a sequence of pairs $(\alpha_{ij}, \beta_{ij})$ and a sequence of values $\delta_i = d_i^{-1/2}$ that leave the array on the right. In addition, intermediate values[4] of δ_i leave the array at the bottom of processor p_{R1}; for example $d_4^{(3)}$ in time step 25. The diagonal processors p_{jj} of the array compute the sequence of pairs $(\alpha_{ij}, \beta_{ij})$ for the entire column j, that is for all rows i with $j < i \leq M$. In addition, these processors compute value $\gamma_{ij} = -\alpha_{ij}\beta_{ij}$, which is used to update elements $d_i^{(n+1)} = (1+\gamma_{ij})d_i^{(n)}$ and $d_j^{(n+1)} = (1+\gamma_{ij})d_j^{(n)}$ of diagonal matrix D. After all updates are applied, one computes $\delta_i = 1/\sqrt{d_i^{(n)}}$. The updates of the diagonal elements are scheduled on either the diagonal or subdiagonal processors of the array.

[3]The systolic array for Givens computation is based on the array proposed by Gentleman and Kung [10]. Bojanczk, Brent, and Kung describe an alternative array for QR factorization [6]

[4]The sequence of n updates of diagonal element d_i is dentoed as: d_i, $d_i^{(1)}$, $d_i^{(2)}$, $d_i^{(3)}$, …, $d_i^{(n)}$, and, finally, compute $\delta_i = 1/\sqrt{d_i^{(n)}}$. Similarly, upper triangular element r_{ij} $(i \leq j)$ evolves from a_{ij} via a sequence of n intermediate values: a_{ij}, $u_{ij}^{(1)}$, $u_{ij}^{(2)}$, …, $u_{ij}^{(n)}$, $r_{ij} = \delta_i u_{ij}^{(n)}$.

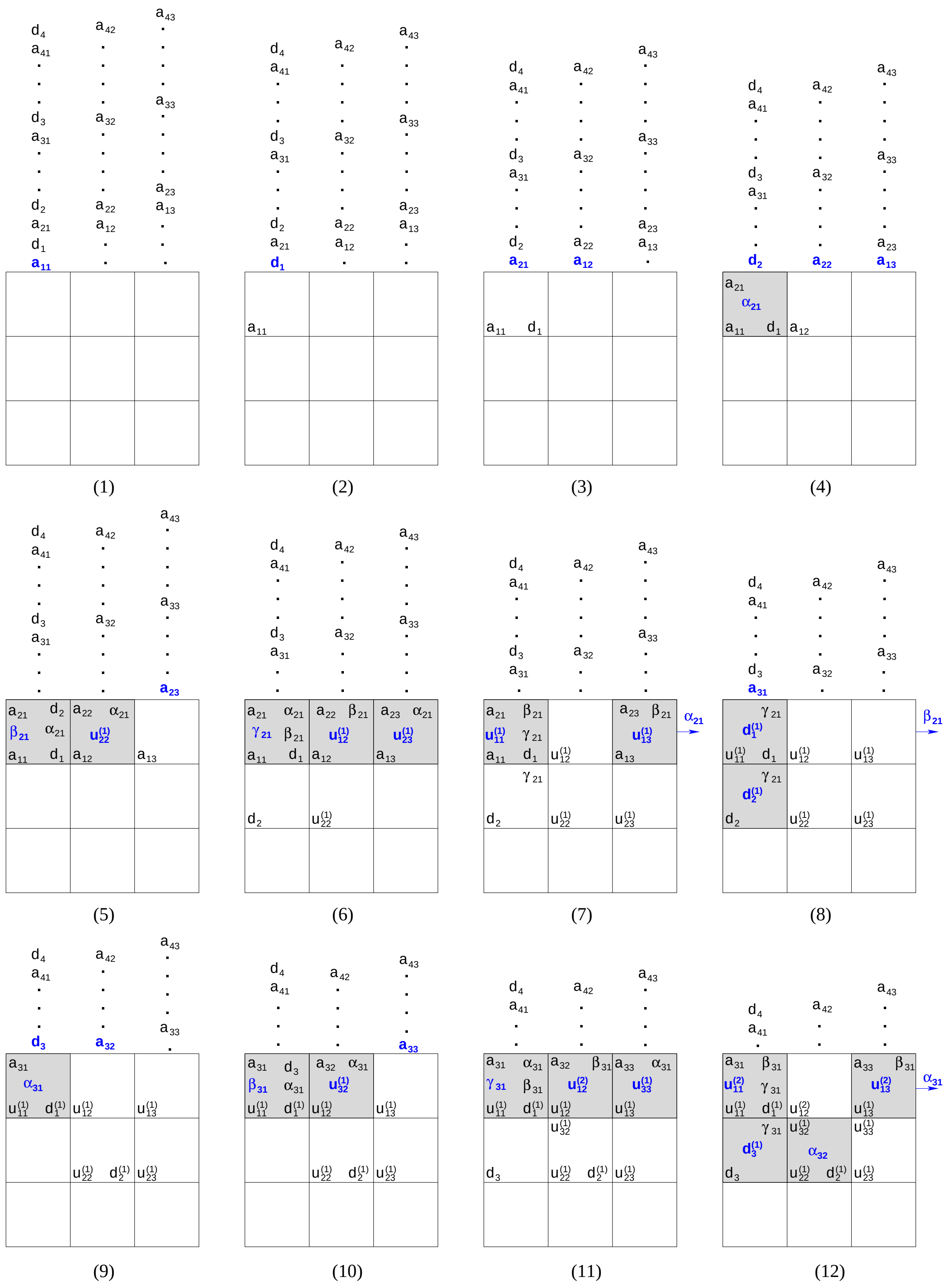


Figure 8-2: Systolic Givens computation for an $M \times R$ matrix, where $M \geq R$. The figure illustrates the triangularization of a 4×3 matrix A, such that $R = D^{-1/2}$ $G(4,3)^T$ $G(4,2)^T$ $G(3,2)^T$ $G(4,1)^T$ $G(3,1)^T$ $G(2,1)^T \cdot A$. The fast Givens transformation $G(i,j)$ is represented by the pair $(\alpha_{ij}, \beta_{ij})$, and $\delta_i = d_i^{-1/2}$. Figure 8-4 shows the pseudocode executed by each processor in the systolic array. [continued in Figure 8-3]

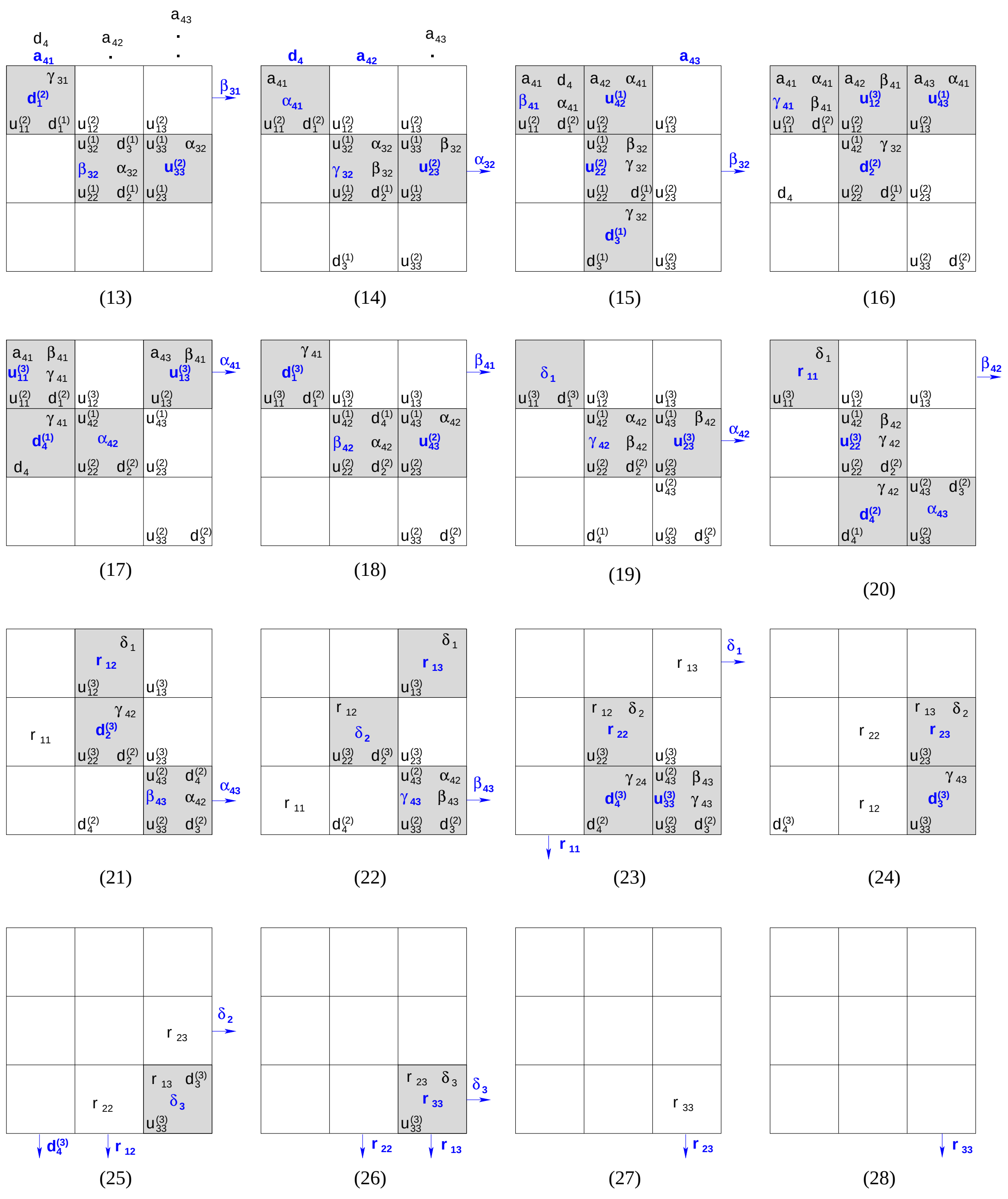


Figure 8-3: Continuation of Figure 8-2.

```
if $i = 1$ and $j = 1$
  $x \leftarrow$Net($north$)
  $d \leftarrow$Net($north$)
  for $m$: 1 .. $M - i$
    $\alpha \leftarrow -$Net($north$)$/x$ route Net($north$) $\rightarrow a$
    $\beta \leftarrow -\alpha \times -d/$Net($north$) route Net($north$) $\rightarrow$Net($south$),
                                                    $\alpha \rightarrow$Net($east$)
    $\gamma \leftarrow -\alpha \times \beta$ route $\beta \rightarrow$Net($east$)
    $x \leftarrow x + \beta \times a$ route $\gamma \rightarrow$Net($south$)
    $d \leftarrow d + d \times \gamma$
  $\delta \leftarrow 1/\sqrt{d}$
  $x \leftarrow x \times \delta$ route $\delta \rightarrow$Net($east$)
  for $n$: 1 .. $i - 1$
    nop route Net($north$) $\rightarrow$ Net($south$)
  Net($south$) $\leftarrow$ $x$

else if $i = R$ and $j = R$
  $x \leftarrow$Net($north$)
  $d \leftarrow$Net($north$)
  for $m$: 1 .. $M - i$
    $\alpha \leftarrow -$Net($north$)$/x$ route Net($north$) $\rightarrow a$
    $\beta \leftarrow -\alpha \times -d/$Net($west$) route Net($west$) $\rightarrow$Net($west$),
                                                    $\alpha \rightarrow$Net($east$)
    $\gamma \leftarrow -\alpha \times \beta$ route $\beta \rightarrow$Net($east$)
    $x \leftarrow x + \beta \times a$ route $\gamma \rightarrow$Net($west$)
    $d \leftarrow d + d \times \gamma$
  $\delta \leftarrow 1/\sqrt{d}$
  $x \leftarrow x \times \delta$ route $\delta \rightarrow$Net($east$)
  for $n$: 1 .. $i - 1$
    nop route Net($north$) $\rightarrow$ Net($south$)
  Net($south$) $\leftarrow$ $x$
```

Figure 8-4: Pseudo code for compute processor p_{ij} executing systolic Givens computation. Here the code is shown for the phase that ouputs intermediate values of d to the bottom of the array. One should note that each processor requires only a bounded amount of storage to hold the values x, d, a, m, α, β, γ, δ, n, i, j, R, and M. [continued in Figure 8-5]

```
else if $i = j$
  $x \leftarrow$Net($north$)
  $d \leftarrow$Net($west$)
  for $m$: 1 .. $M - i$
    $\alpha \leftarrow -$Net($north$)$/x$ route Net($north$) $\rightarrow a$
    $\beta \leftarrow -\alpha \times -d/$Net($west$) route Net($west$) $\rightarrow$Net($south$),
                                                         $\alpha \rightarrow$Net($east$)
    $\gamma \leftarrow -\alpha \times \beta$ route $\beta \rightarrow$Net($east$)
    $x \leftarrow x + \beta \times a$ route $\gamma \rightarrow$Net($south$)
    $d \leftarrow d + d \times \gamma$
  $\delta \leftarrow 1/\sqrt{d}$
  $x \leftarrow x \times \delta$ route $\delta \rightarrow$Net($east$)
  for $n$: 1 .. $i - 1$
    nop route Net($north$) $\rightarrow$ Net($south$)
  Net($south$) $\leftarrow$ $x$

else if $j > i$
  $x \leftarrow$Net($north$)
  for $m$: 1 .. $M - i$
    Net($south$) $\leftarrow x\times$Net($west$)+Net($north$) route Net($west$) $\rightarrow$Net($east$),
                                                         Net($north$) $\rightarrow a$
    $x \leftarrow a\times$Net($west$) $+ x$ route Net($west$) $\rightarrow$Net($east$)
  for $n$: 1 .. $i - 1$
    nop route Net($north$) $\rightarrow$ Net($south$)
  Net($south$) $\leftarrow$ $x/$Net($west$) route Net($west$) $\rightarrow$Net($east$)

else if $i = R$ and $j = R - 1$
    $d \leftarrow$Net($north$)
    Net($east$) $\leftarrow d + d\times$Net($north$)
  for $m$: 2 .. $M - i$
    $d \leftarrow$Net($north$)
    Net($east$) $\leftarrow d + d\times$Net($north$)
    nop route Net($east$) $\rightarrow$Net($west$)
  for $n$: 1 .. $i - 1$
    nop route Net($north$) $\rightarrow$ Net($south$)
```

Figure 8-5: Continuation of Figure 8-4. [continued in Figure 8-6]

```
else if $i = j + 1$
  for $m$: 1 .. $M - i$
    $d \leftarrow$Net($north$)
    Net($east$) $\leftarrow d + d\times$Net($north$)
  for $n$: 1 .. $i - 1$
    nop route Net($north$) $\rightarrow$ Net($south$)

else if $i \neq R$ and $i > j$
  for $n$: 1 .. $i - 1$
    nop route Net($north$) $\rightarrow$ Net($south$)

else if $i = R$ and $j = 1$
  for $n$: 1 .. $N - R$
    nop route Net($east$) $\rightarrow$ Net($south$)
  for $n$: 1 .. $i - 1$
    nop route Net($north$) $\rightarrow$ Net($south$)

else if $i = R$
  for $n$: 1 .. $N - R$
    nop route Net($east$) $\rightarrow$ Net($west$)
  for $n$: 1 .. $i - 1$
    nop route Net($north$) $\rightarrow$ Net($south$)
```

Figure 8-6: Continuation of Figure 8-5.

The systolic Givens computation involves a relatively large number of operations even for the small 4×3 example in Figures 8-2 and 8-3. Therefore, rather then discussing the computation of a particular element, the data flow through the array is described at a higher level. As mentioned before, the processors on the diagonal of the array are responsible for computing the fast Givens transformations. In particular, processor p_{11} computes the transformations $G(2,1)$ during time steps 4–8, $G(3,1)$ during time steps 9–13, and $G(4,1)$ during time steps 14–18. For example, transformation $G(2,1)$ involves computing α_{21} in time step 4, β_{21} in time step 5, γ_{21} in time step 6, and updating the diagonal elements d_1 and d_2 during time step 8 on the diagonal and subdiagonal processors p_{11} and p_{21}. Analogously, the fast Givens transformations $G(3,2)$, and $G(4,2)$ are computed on processor p_{22} with support of p_{32} during time steps 12–21, and $G(4,3)$ on processor p_{33} during time steps 20–25.

The systolic Givens computation in Figures 8-2 and 8-3 produces an upper triangular 3×3 matrix by computing the premultiplications with the fast Givens transformations. All updates due to premultiplications occur on the upper triangular processors of the array. They generate the elements r_{ij} $(i \le j)$ via a sequence of intermediate values: a_{ij}, $u_{ij}^{(1)}$, $u_{ij}^{(2)}$, …, $u_{ij}^{(n)}$, $r_{ij} = \delta_i u_{ij}^{(n)}$. Recall that the updates of diagonal element δ_i involve multiplications with $(1+\gamma_{ij})$ for $i > j$ and with $(1+\gamma_{ji})$ for $i < j$. The systolic Givens computation according to Equation 8.3 will therefore generate intermediate values of δ_i, that require further updates when computing Equation 8.5. In Figure 8-3, only one intermediate value, $d_4^{(3)}$, is produced, which leaves the array at the bottom of p_{31} at time step 25. It will enter the array again, as explained during the discussion of Figure 8-12 below.

Figures 8-7 and 8-8 illustrate the systolic premultiplication, which updates matrix A according to Equation 8.4. Figure 8-5 shows the pseudocode executed by each processor performing the systolic premultiplication. This update uses the pair-representation of the Givens transformations, so that matrix Q_1^T does not have to be computed explicitly. The systolic array produces the $R \times R$ matrix R_{12} and the $(M - R) \times R$ matrix $(A'_{22}\ A'_{32})^T$ of Equation 8.4. Columns of matrix A enter the array at the top, and the pairs $(\alpha_{ij}, \beta_{ij})$ of fast Givens transformation $G(i,j)$ as well as the diagonal elements of matrix $\hat{D}^{-1/2}$ enter the array from the right. Processor p_{ij} computes element r_{ij} of matrix R_{12}. The elements of matrices R_{12} and $(A'_{22}\ A'_{32})^T$ leave the array at the bottom such that the values of $(A'_{22}\ A'_{32})^T$ precede those of R_{12}.

Compared to the systolic Givens computation, the data flow through the array in Figures 8-7 and 8-8 is relatively straightforward. The figures illustrate the systolic premultiplication of a 4×3 matrix A. According to Equation 8.4, the premultiplication for this particular example is:

$$\begin{pmatrix} R_{12} \\ A'_{22} \end{pmatrix} = \hat{D}^{-1/2}G(4,3)^TG(4,2)^TG(3,2)^TG(4,1)^TG(3,1)^TG(2,1)^T \begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix},$$

where the multiplications are executed from right to left. For example, consider the computation of element r_{11}. Its value is determined by the premultiplications with $G(2,1)$, $G(3,1)$, and $G(4,1)$, generating a sequence of intermediate values a_{11}, $u_{11}^{(1)}$, $u_{11}^{(2)}$, $u_{11}^{(3)}$, and finally r_{11}: $u_{11}^{(1)} = a_{11} + \beta_{21}\cdot a_{21}$, $u_{11}^{(2)} = u_{11}^{(1)} + \beta_{31}\cdot a_{31}$, $u_{11}^{(3)} = u_{11}^{(2)} + \beta_{41}\cdot a_{41}$, $r_{11} = \delta_1 \cdot u_{11}^{(3)}$. In Figures 8-7 and 8-8, each of these updates is computed by processor p_{11}. The values of the first column of A enter p_{11} from the top. The (α,β)-pairs associated with the fast Givens transformations $G(2,1)$, $G(3,1)$, and $G(4,1)$ enter p_{11} from the right. Values a_{21} and β_{21} are available for the first update at p_{11} during time step 6, resulting in $u_{11}^{(1)}$. The second update occurs at time step 8, and the third at time step 10. At time step 11, diagonal element δ_1 arrives at processor p_{11}, resulting in the computation of r_{11}. Thereafter, r_{11} travels downwards through the array, one processor per time step, until it leaves the array at the bottom of processor p_{31} at time step 14.

The computations of elements a'_{ij} for $i > R$ are analogous to those of the r_{ij}. For example, the computation of a'_{41} produces the sequence of intermediate values: a_{41}, $a_{41}^{(1)} = a_{41} + \alpha_{41}a_{11}$, $a_{41}^{(2)} = a_{41}^{(1)} + \alpha_{42}u_{21}^{(1)}$, $a'_{41} = a_{41}^{(2)} + \alpha_{43}u_{31}^{(2)}$. The corresponding updates occur at time step 9 on processor p_{11}, time step 10 on processor p_{21}, and time step 11 on processor p_{31}. Element a'_{41} leaves the array at the bottom of processor p_{31} at time step 12.

Figures 8-10 and 8-11 illustrate the systolic postmultiplication, which computes

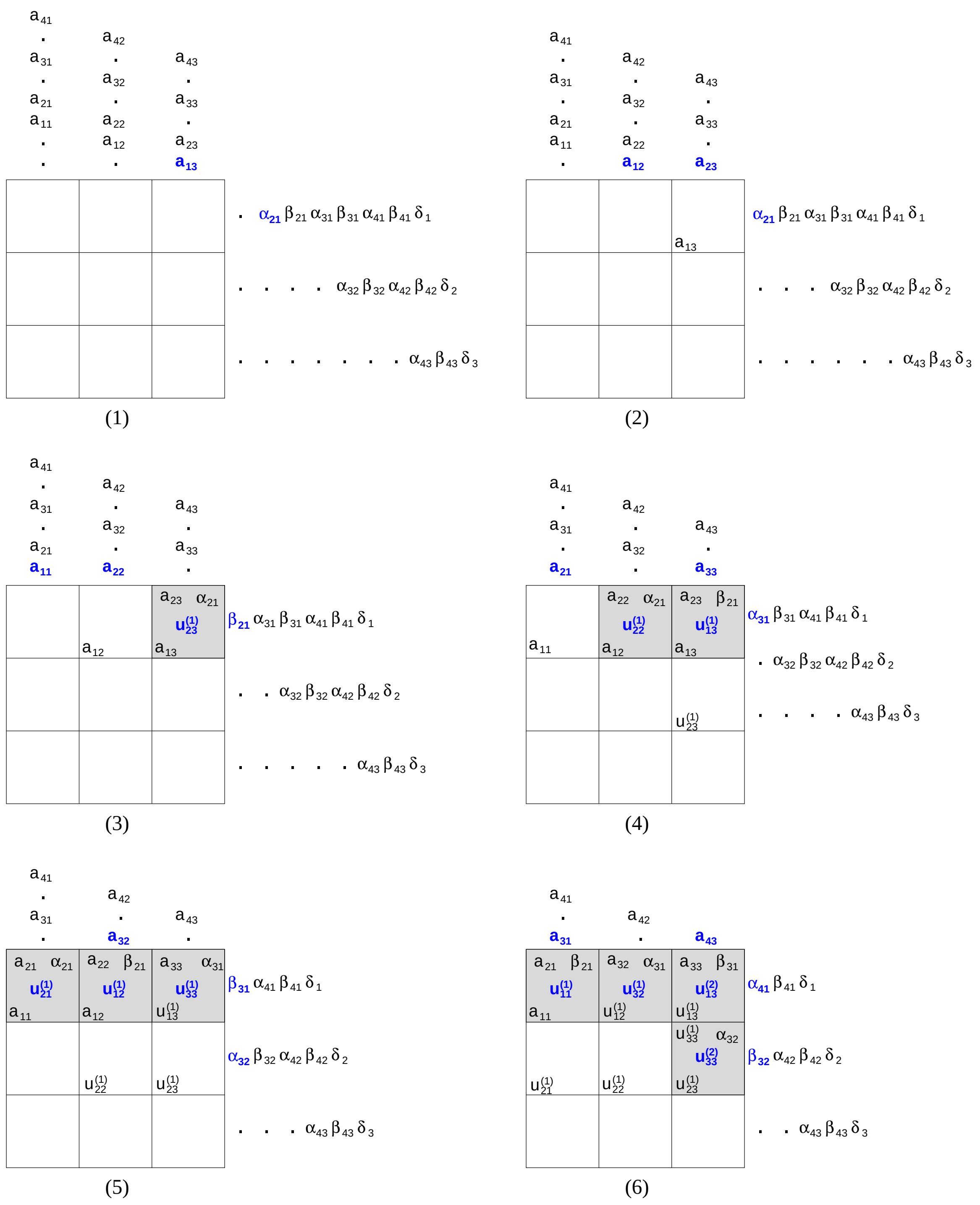


Figure 8-7: Systolic premultiplication for computing R and updating A according to Equation 8.4. Figure 8-9 shows the pseudocode executed by each processor in the array. [continued in Figure 8-8]

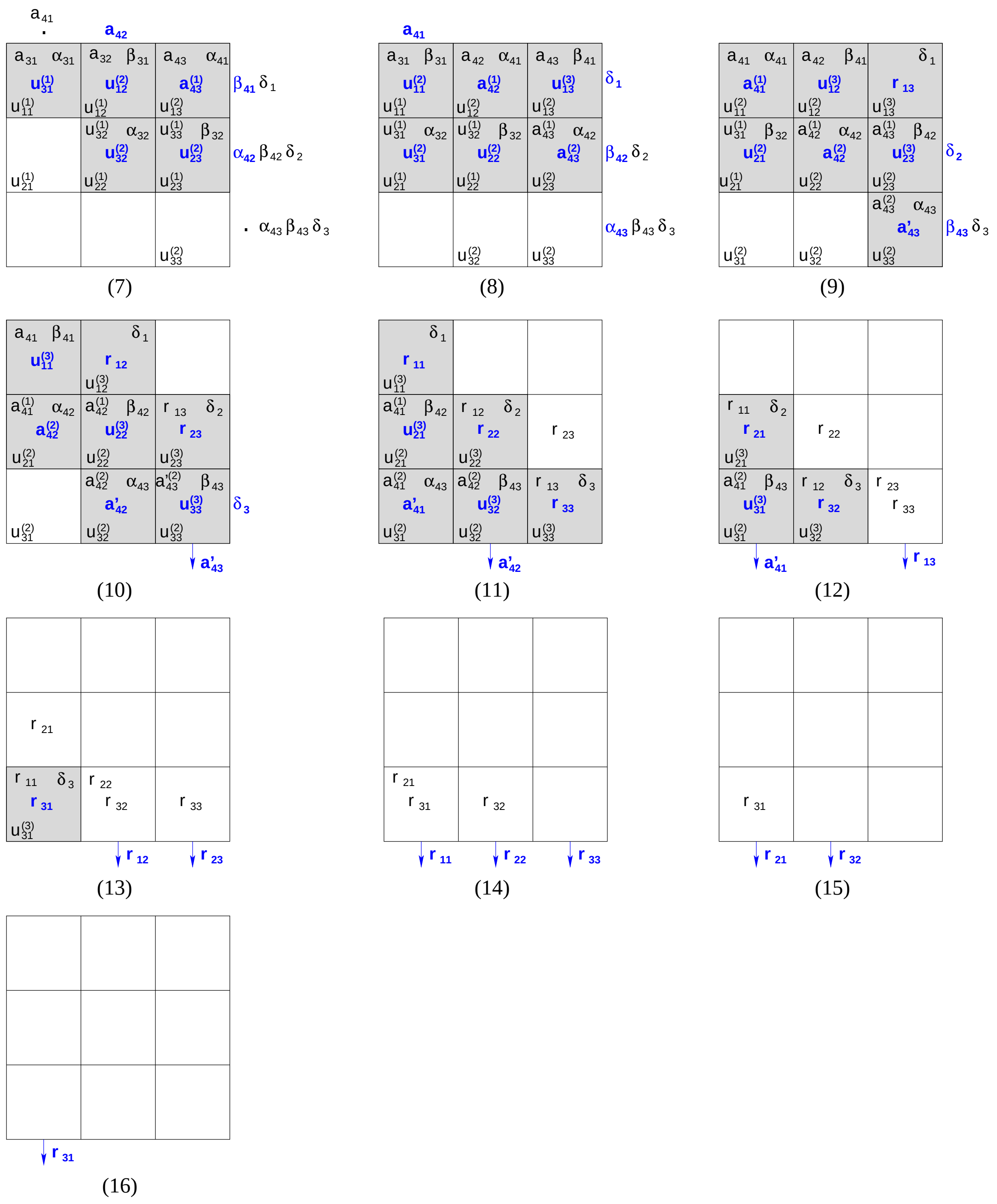


Figure 8-8: Continuation of Figure 8-7.

```
$x \leftarrow$ Net($north$)

**if** $j \neq 1$
  **for** $m$: 1 .. $M - i$
    Net($south$) $\leftarrow x \times$Net($east$)+Net($north$), **route** Net($east$) $\rightarrow$Net($west$),
                                                  Net($north$) $\rightarrow a$
    $x \leftarrow a \times$Net($east$) $+ x$, **route** Net($east$) $\rightarrow$Net($west$)
  **for** $n$: 1 .. $i - 1$
    nop, **route** Net($north$) $\rightarrow$ Net($south$)
  Net($south$) $\leftarrow$ $x$/Net($east$), **route** Net($east$) $\rightarrow$Net($west$)

**else**
  **for** $m$: 1 .. $M - i$
    Net($south$) $\leftarrow x \times$Net($east$)+Net($north$), **route** Net($north$) $\rightarrow a$
    $x \leftarrow a \times$Net($east$) $+ x$
  **for** $n$: 1 .. $i - 1$
    nop, **route** Net($north$) $\rightarrow$ Net($south$)
  Net($south$) $\leftarrow$ $x$/Net($east$)
```

Figure 8-9: Pseudocode for compute processor p_{ij} executing systolic premultiplication. Note that each processor requires only a bounded amound of storage to hold the values of m, i, j, a, x, and M.

matrix Q of the QR factorization according to Equation 8.8 by postmultiplying the sequence of fast Givens transformations into the identity matrix. The pseudocode in Figure 8-9 can also be used to compute a systolic postmultiplication. In this case, each column of processors would compute one row of output values. Here, intermediate values of G are denoted as Q' according to

$$\begin{pmatrix} Q_{11} & Q'_{12} & Q'_{13} \end{pmatrix} = \begin{pmatrix} I & 0 & 0 \end{pmatrix} \cdot \left(\prod_{j=1}^{R} \prod_{i=j+1}^{M} G(i,j) \right) \cdot \hat{D}^{-1/2}$$

where Q_{11} is an $R \times R$ submatrix of Q, Q'_{12} and Q'_{13} are $R \times R$ matrices of intermediate values, and I is an $R \times R$ identity matrix. The $R \times R$ array of compute processors in Figures 8-10 and 8-11 generates an $R \times M$ block of rows at a time, where $M \geq R$. Matrix $G = (g_{ij})$, which is initially the $M \times M$ identity matrix, enters the processor array at the top. The fast Givens transformations are represented by their (α, β)-pairs, and enter the array from the right. The diagonal elements of matrix $\hat{D}^{-1/2}$ enter the array from the right following the fast Givens transformations. Matrices Q_{11}, Q'_{12} and Q'_{13} leave the array at the bottom, such that processor p_{Ri} emits row i of matrix $(Q_{11}\ Q'_{12}\ Q'_{13})$, and the values of Q'_{12} and Q'_{13} precede those of Q_{11}.

Processor p_{ij} of the array computes element q_{ij} of Q_{11}. For example, consider the computation of element q_{11}, which results from postmultiplications with $G(2,1)$, $G(3,1)$, and $G(4,1)$. Starting with g_{11}, processor p_{11} generates the sequence of intermediate values: $g_{11}^{(1)} = g_{11} + \beta_{21} g_{12}$, $g_{11}^{(2)} = g_{11}^{(1)} + \beta_{31} g_{13}$, $g_{11}^{(3)} = g_{11}^{(2)} + \beta_{41} g_{14}$, and finally $q_{11} = \delta_1 g_{11}^{(3)}$. The first update occurs on processor p_{11} at time step 6, the second at time step 8, the third at time step 10, and the computation of q_{11} at time step 11. The intermediate values $g_{ij}^{(R)}$ for $j > R$ are computed before the elements of Q_{11}. In Figure 8-11, these are the values $g_{14}^{(3)}$, $g_{24}^{(3)}$, and $g_{34}^{(3)}$. One should note that the number of updates involving zero-elements of the initial matrix $G = I$ is asymptotically insignificant compared to the total number of updates. Hence, one may leave the updates involving zero-elements in the computation to keep the systolic array as simple as possible.

One is now ready to compose the three systolic algorithms for fast Givens com-

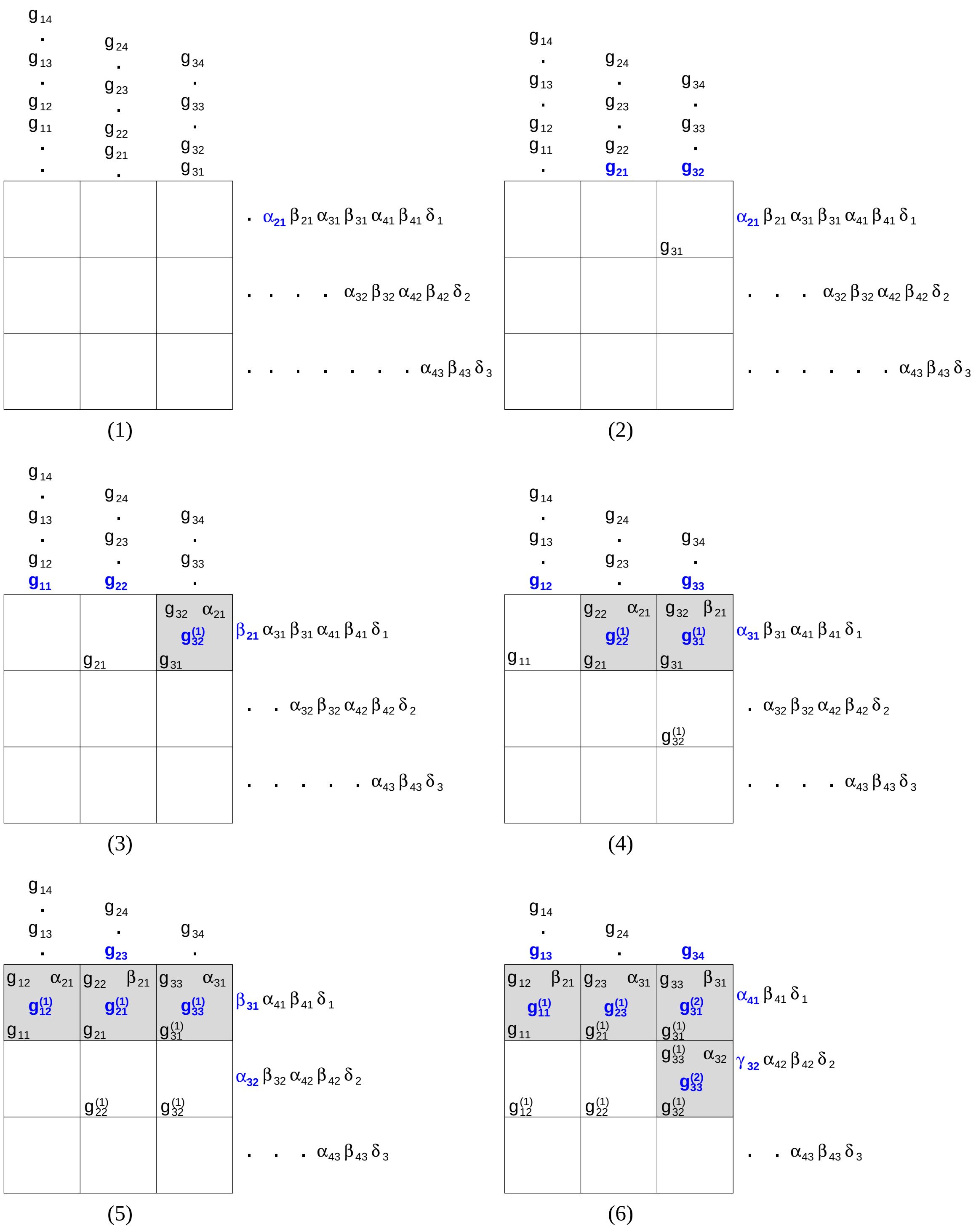


Figure 8-10: Systolic postmultiplication for computing an $R \times M$ block of $Q = IG(2,1)G(3,1)G(4,1)G(3,2)G(4,2)G(4,3)D^{-1/2}$. [continued in Figure 8-11]

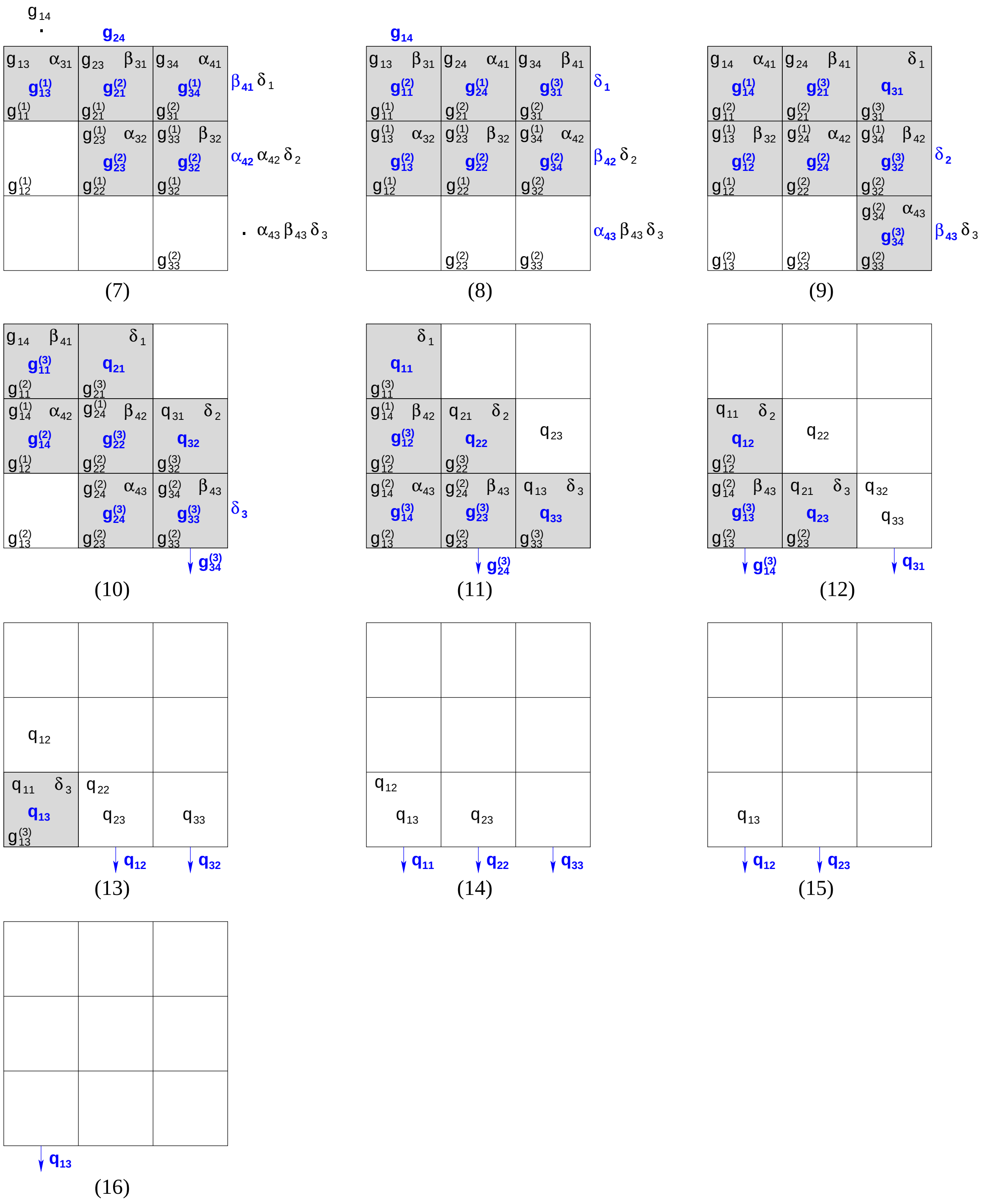


Figure 8-11: Continuation of Figure 8-10.

putation, premultiplication, and postmultiplication, and discuss the decoupled data flow of the complete QR factorization. Figure 8-12 shows the twelve systolic computations needed to compute both Q and R of an $M \times N$ matrix A for $M = 3R$ and $N = 2R$ on an $R \times R$ array of compute processors. In addition, $3R$ memory processors are required on the periphery of the compute array. The computation uses two fast Givens computations to compute R_{11}, R_{22}, and the corresponding fast Givens transformations in phase 1 according to Equation 8.3 and phase 3 according to Equation 8.5, respectively. Phase 2 implements the premultiplication according to Equation 8.4. The remaining phases use the postmultiplication to compute matrix Q. In particular, matrix Q_1 of Equation 8.6 is computed during phases 4–6, one $R \times M$ row block per phase, and matrix Q is computed during phases 7–12.

The fast Givens computation of phase 3 is reflected about the horizontal axis, when compared to the presentation in Figures 8-2 and 8-3. This reflected version of the systolic algorithm allows one to stream the results of the premultiplication from phase 2 immediately back into the array. Similarly, the computation assumes reflected versions of the postmultiplication during phases 7–9. One may note that the computation of R is finished after phase 3. Thus, for linear system solvers that do not require knowledge of Q, one could stop the computation after these three phases. Phases 7–12 for the computation of Q deserve further discussion, because this implementation deviates slightly from Equations 8.6–8.8. Rather than computing Q_2 explicitly, it is more efficient to postmultiply Q_1 with the corresponding fast Givens transformations directly. However, these postmultiplications apply to the right-most $M \times (M - R)$ submatrix of Q_1 only, as shown in phases 7–9. Finally, the right-most column block $(Q''_{13}\ Q''_{23}\ Q''_{33})^T$ of the intermediate matrix must be multiplied by $\tilde{D}^{-1/2}$ according to Equation 8.7. This multiplication is implemented in phases 10–12. Matrix Q is now available in parts on the memory processors at the top and in parts at the bottom of the machine.

Analogous to other stream algorithms, the partitioning due to Equations 8.3–8.7 produces a set of heterogeneous subproblems. One should reduce the QR factorization of size $M \times N$ until the subproblems can be computed on an $R \times R$ array of compute

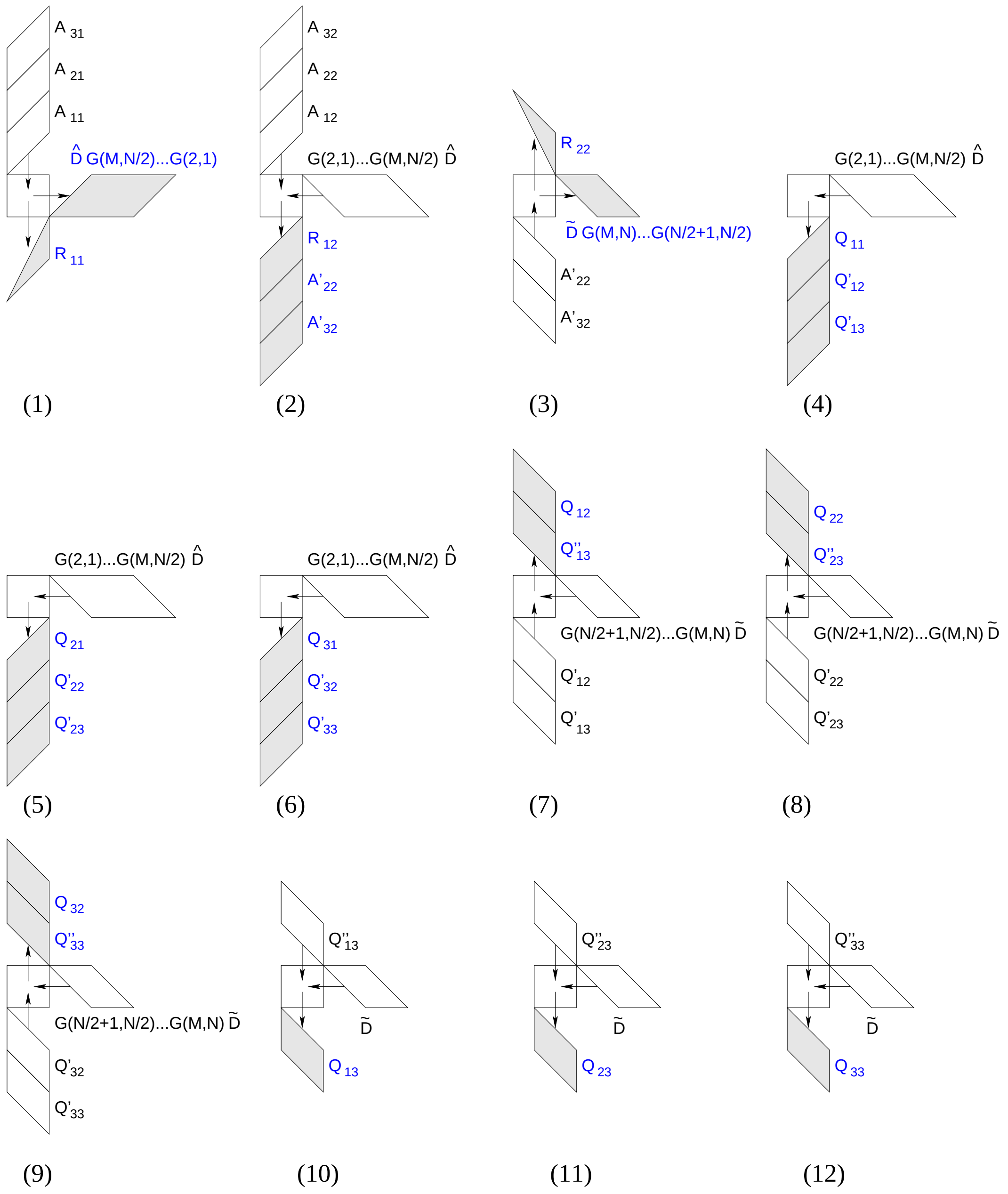


Figure 8-12: Phases of a stream-structured QR factorization on an $R \times R$ array of compute processors, with $M = 3R$ and $N = 2R$. In phase 1, one triangularizes the left-most $N/2$ columns of A, producing R_{11} and a set of fast Givens transformations as expressed in Equation 8.3. In phase 2, the Givens transformations are applied to the right-most $N/2$ columns of A yielding R_{12}, A'_{22}, and A'_{32} according to Equation 8.4. In phase 3, one triangularizes the right-most columns of A according to Equation 8.5. Phases 4-6 compute Q_1 according to Equation 8.6, while phases 7-12 compute $Q = Q_1Q_2$ according to Equation 8.8. Note that phases 3 and 7–12 can be modified without loss of efficiency, so that matrix Q is not distributed across the top and bottom rows of memory processors, but would be available on one side only.

processors. Figure 8-12 represents the data movement when computing the systolic subproblems of Equations 8.3–8.8. This structure requires R memory processors at the top of the array to buffer the columns of A and the intermediate results, another R at the bottom of the array for the rows of Q and columns of R, and R more to the right of the array for storing Givens transformations. Thus, the decoupled systolic QR factorization requires $P = R^2$ compute processors and $M = 3R$ memory processors. Therefore, $M = o(P)$ and thus the QR factorization is decoupling efficient.

8.3 Efficiency Analysis

In the following, the efficiencies of computing R and Q of an $M \times N$ matrix A are analyzed separately, because the computation of Q is optional. To handle the general case where $M \geq N$, two σ-values $\sigma_M = M/R$ and $\sigma_N = N/R$ are introduced, where network size is again represented by R. One may approximate the number of multiply-and-add operations for computing matrix R by means of fast Givens transformations as $C_R(M,N) \approx N^2(M - N/3)$. This approximation neglects an asymptotically insignificant quadratic term that includes division operations and a linear term including square-root operations.

The analysis begins by calculating the number of time steps required by the computation of matrix R. One should apply the methodology used for the lower-triangular solver and the LU factorization, and partition the problem recursively until matrix A consists of $\sigma_M \times \sigma_N$ blocks, each of size $R \times R$. Equation 8.9 illustrates the partitioning when $\sigma_M = 5$ and $\sigma_N = 4$.

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \\ A_{51} & A_{52} & A_{53} & A_{54} \end{pmatrix} = Q \begin{pmatrix} R_{11} & R_{12} & R_{13} & R_{14} \\ 0 & R_{22} & R_{23} & R_{24} \\ 0 & 0 & R_{33} & R_{34} \\ 0 & 0 & 0 & R_{44} \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{8.9}$$

To facilitate the understanding of the efficiency analysis, the block-iterative schedule is explained in detail. The partitioning in Equations 8.3–8.5 extends to the 5×4 case as follows. The factorization sweeps across pivot blocks from top left to bottom right. The computation of each pivot block includes a triangularization step and an update step. First, one annihilates all lower-triangular elements in the pivot column using a systolic fast Givens computation. Then, one may update the column blocks to the right of the pivot column by means of premultiplications, cf. phases 1 and 2 in Figure 8-12. In the example of Equation 8.9, one first annihilates all lower-triangular elements in column block A_{i1}. Second, one must premultiply the 5×3 block matrix consisting of column blocks $(A_{i2}\ A_{i3}\ A_{i4})$ for $1 \leq i \leq 5$. One applies the systolic premultiplication once to each of these column blocks separately and then proceeds with triangularizing column block A_{i2}. The subsequent premultiplication effects the matrix $(A_{i3}\ A_{i4})$ for $2 \leq i \leq 5$, that is excluding the top row. Similarly, after triangularizing column block A_{i3}, the subsequent premultiplication effects matrix $(A_{44}\ A_{54})^T$ only. Figure 8-13 shows the areas of matrix A that are effected by the fast Givens computation (a) and the premultiplication (b) when handling column block i.

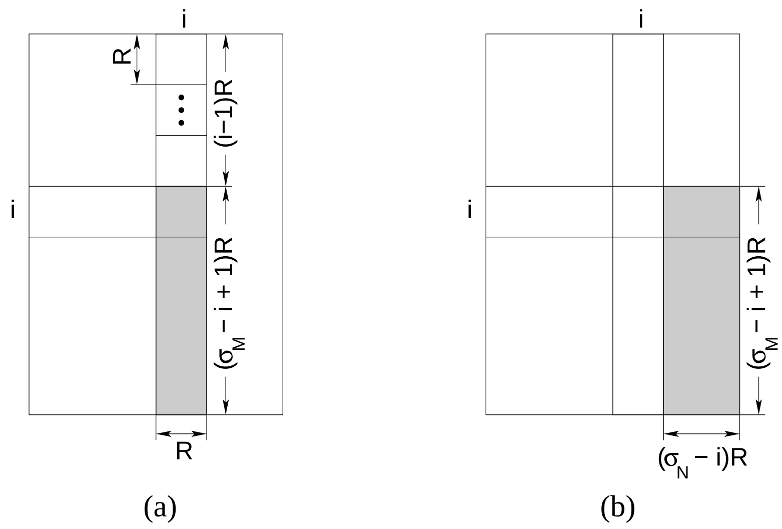


Figure 8-13: (a) The fast Givens computation of Equation 8.3 (or Equation 8.5) for column block i effects the bottom $(\sigma_M - i + 1)R$ rows. (b) After triangularizing column block i, one updates the $(\sigma_M - i + 1)R \times (\sigma_N - i)R$ submatrix of A according to Equation 8.4.

Now one may consider the general case where A is a $\sigma_M \times \sigma_N$ matrix. The behavior of the systolic fast Givens computation shown in Figures 8-2 and 8-3 is as follows. The critical path is determined by the computations of the processors on the diagonal of the array. Processor p_{kk} requires 5 time steps to compute α, β, γ, and intermediate values of d_k and u_{kk} for each Givens transformation. Thus, the fast Givens transformation of column block i uses $5(\sigma_M - i + 1)R$ time steps. In addition, starting up the pipeline takes $5R$ time steps and draining $10R$ time steps, resulting in a total of $5(\sigma_M - i + 1)R + 15R$ time steps.

The number of time steps for the premultiplications associated with column block i can be counted as follows. The systolic premultiplication requires 2 time steps per output value according to Figures 8-7 and 8-8. For $(\sigma_N - i)$ column blocks and with $(\sigma_M - i + 1)$ row blocks each, the premultiplication takes $(\sigma_N - i) \cdot 2(\sigma_M - i + 1)R$ time steps. In addition, starting the pipeline requires $2R$ time steps and draining $4R$ time steps for a total of $2(\sigma_N - i)(\sigma_M - i + 1)R + 6R$ time steps.

The number of time steps for computing upper-triangular matrix R of the stream-structured QR factorization is summed up as follows. For a $\sigma_M \times \sigma_N$ block matrix A with block size $R \times R$ on a network of size R, where the postmultiplication can overlap the preceding fast Givens computation by R time steps, one has

$$\begin{aligned}
T_r(\sigma_M, \sigma_N, R) \;&\approx\; \sum_{i=1}^{\sigma_N} 5(\sigma_M - i + 1)R + 15R \\
&\quad + \sum_{i=1}^{\sigma_N - 1} 2(\sigma_N - i)(\sigma_M - i + 1)R + 6R \\
&\quad - \sum_{i=1}^{\sigma_N - 1} R \\
&= R(\sigma_N^2 \sigma_M - \frac{1}{3}\sigma_N^3 + 4\sigma_M \sigma_N - \frac{3}{2}\sigma_N^2 + \frac{101}{6}\sigma_N - 5).
\end{aligned}$$

Using a network of size R, $P = R^2$ compute processors, and $M = 3R$ memory processors, the floating-point efficiency of the computation of upper triangular matrix R of the QR factorization is

$$E_r(\sigma_M, \sigma_N, R) \;\approx\; \frac{\sigma_N^2 \sigma_M - \frac{1}{3}\sigma_N^3}{\sigma_N^2 \sigma_M - \frac{1}{3}\sigma_N^3 + 4\sigma_M \sigma_N - \frac{3}{2}\sigma_N^2 + \frac{101}{6}\sigma_N - 5} \cdot \frac{R}{R+3}.$$

Asymptotically, that is for large network size R, σ_M, and σ_N, the compute efficiency approaches the optimal value of 100 %. As for the triangular-solver and LU factorization, when $\sigma_M, \sigma_N \gg 1$, $E_r(R) \approx R/(R+3)$, and the algorithm achieves more than 90 % efficiency for $R \geq 27$.

Now consider the efficiency analysis of the computation of matrix Q of the QR factorization. The number of multiply-and-add operations is $C_Q(M, N) = 2M^2N - MN - MN^2$. The most efficient schedule uses a block-iterative approach to overlap and pipeline the systolic postmultiplications. The problem is partitioned recursively until Q is a $\sigma_M \times \sigma_M$ block matrix, and each block is a $R \times R$ matrix. The postmultiplications associated with column block i of Q also effect the all column blocks $i+1, \ldots, \sigma_M$ to the right of i, as shown in Figure 8-14. One applies the systolic postmultiplication to individual row blocks. Thus, the computation of column block i applies σ_M systolic postmultiplications to a row block of size $R \times (\sigma_M - i + 1)R$. According to Figures 8-10 and 8-11, the computation of each output value requires 2 time steps. Therefore, the number of time steps for postmultiplying σ_M row blocks associated with column block i is $2R\sigma_M(\sigma_M - i + 1)$. With a startup time for the systolic postmultiplication of $2R$ time steps, and a drainage time of $4R$ time steps, the number of time steps for column block i is $2R\sigma_M(\sigma_M - i + 1) + 6R$.

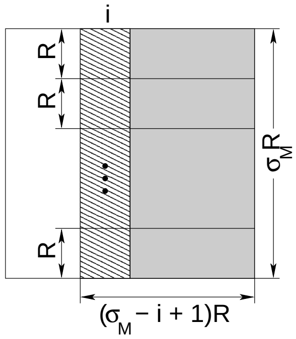


Figure 8-14: One computes column block i of $M \times M$ matrix Q by partitioning the computation into σ_M row blocks, each with R rows and $(\sigma_M - i + 1)R$ columns. Each row block is computed using a systolic postmultiplication. After the postmultiplication, the hatched area of the matrix holds the final values of Q, while the shaded area comprises intermediate values.

One obtains the number of time steps for computing matrix Q of the QR factorization of an $M \times N$ matrix A by summing up the number of time steps for each update of matrix Q. Matrix Q is updated once for each set of Givens transformations produced by the systolic Givens computation array, and there are σ_N sets of Givens transformations. One can save $2R$ time steps per column block by overlapping consecutive computations.

$$\begin{aligned} T_q(\sigma_M, \sigma_N, R) &= \sum_{i=1}^{\sigma_N} (2R\sigma_M(\sigma_M - i + 1) + 6R) - \sum_{i=1}^{\sigma_N - 1} 2R \\ &= R\left(2\sigma_M^2\sigma_N - \sigma_N^2\sigma_M + \sigma_N\sigma_M + 4\sigma_N + 2\right) \end{aligned}$$

Using a network of size R, $P = R^2$ compute processors, and $M = 3R$ memory processors, the floating-point efficiency of computing matrix Q of the QR-factorization is

$$E_q(\sigma_M, \sigma_N, R) = \frac{2\sigma_M^2\sigma_N - \sigma_N^2\sigma_M - \sigma_M\sigma_N}{2\sigma_M^2\sigma_N - \sigma_N^2\sigma_M + \sigma_N\sigma_M + 4\sigma_N + 2} \cdot \frac{R}{R+3}.$$

Asymptotically, that is for large network sizes R, σ_M, and σ_N, the compute efficiency of computing matrix Q approaches the optimal value of 100 %. When $\sigma_M, \sigma_N \gg 1$, $E_q(R) \approx R/(R+3)$, and one achieves more than 90 % efficiency for $R \geq 27$.

The number of time steps and efficiency of the entire QR factorization involves the computation of both R and subsequently Q. The number of multiply-and-add operations is $C(M, N) = C_R(M, N) + C_Q(M, N) \approx 2M^2N - N^3/3$. The number of time steps for computing R and Q is the sum of the time steps for the individual computations:

$$T_{qr}(\sigma_M, \sigma_N, R) \approx R\left(2\sigma_M^2\sigma_N - \frac{1}{3}\sigma_N^3 + 5\sigma_M\sigma_N - \frac{3}{2}\sigma_N^2 + \frac{125}{6}\sigma_N - 3\right).$$

Using a network of size R, $P = R^2$ compute processors, and $M = 3R$ memory processors, the floating-point efficiency of the stream-structured QR factorization is

$$E_{qr}(\sigma_M, \sigma_N, R) \approx \frac{2\sigma_M^2\sigma_N - \frac{1}{3}\sigma_N^3}{2\sigma_M^2\sigma_N - \frac{1}{3}\sigma_N^3 + 5\sigma_M\sigma_N - \frac{3}{2}\sigma_N^2 + \frac{125}{6}\sigma_N - 3} \cdot \frac{R}{R+3}.$$

For $\sigma_M = \sigma_N = 1$, the problem reduces to a single systolic QR factorization, and one obtains a compute efficiency of

$$E_{qr}(\sigma_M = 1, \sigma_N = 1, R) \approx \frac{5}{69} \cdot \frac{R}{R+3}.$$

The expressions for execution time and compute efficiency are easier to comprehend when considering a square matrix A of dimension $N \times N$. Then, $\sigma_M = \sigma_N$, and one can express the execution time and efficiency as a function of $\sigma = \sigma_M = \sigma_N$ and R:

$$T_{qr}(\sigma, R) \approx R\left(\frac{5}{3}\sigma^3 + \frac{7}{2}\sigma^2 + \frac{125}{6}\sigma - 3\right)$$

and

$$E_{qr}(\sigma, R) \approx \frac{\sigma^3}{\sigma^3 + \frac{21}{10}\sigma^2 + \frac{25}{2}\sigma - \frac{9}{5}} \cdot \frac{R}{R+3}.$$

Note that for a fixed σ, the QR factorization of an $N \times N$ matrix requires $T(N) = (\frac{5}{3}\sigma^2 + \frac{7}{2}\sigma + \frac{125}{6} - 3/\sigma)N = \Theta(N)$ time steps with $(N/\sigma)^2$ compute processors.

Typically, when using fast Givens transformations to perform a QR factorization, more than one type of transformation is used. This is done to improve the numerical stability of the algorithm. The above treatment has not shown this method for QR factorization because it is less efficient than the stream-structured QR factorization presented here. However, one may analyze the efficiency of this second form of computation which has superior numerical properties, but is less efficient. The cost of this alternate computation on an $M \times N$ matrix is the same, $C(M, N) \approx 2M^2N - \frac{1}{3}N^3$. Furthermore, the basic structure of all the systolic phases remain the same. However, the number of time steps required for a more stable QR factorization includes the conditional statement that checks for the type of Givens transformation. This check must be done every time a processor applies a Givens transformation and the type of the transformation must be sent through the compute array along with the α and β values. Thus, the number of time steps required per input element for both the systolic postmultiplication and the systolic premultiplication increases from two to three. However, the additional time step is not used for executing a multiply-and-add instruction, but for checking the type of the transformation. Thus, the number of time steps required to execute such a QR factorization becomes:

$$\hat{T}_{qr}(\sigma_M, \sigma_N, R) = \frac{3}{2} T_{qr}(\sigma_M, \sigma_N, R).$$

Using the same sized network, this algorithm has an efficiency of

$$\hat{E}_{qr}(\sigma_M, \sigma_N, R) = \frac{2}{3} E_{qr}(\sigma_M, \sigma_N, R). \tag{8.10}$$

Thus, when using this alternate form of QR factorization, the asymptotic efficiency is bounded by approximately 66 %.

Chapter 9

Convolution

The convolution of vector a of length M with vector w of length N produces an output vector b of length $M+N-1$. Without loss of generality, assume that $M \geq N$. Element k of b is given by

$$b_k \;=\; \sum_{i+j=k+1} a_i \cdot w_j \tag{9.1}$$

where

$$\begin{aligned} 1 \leq \; & k & \leq M+N-1 \\ 1 \leq \; & i & \leq M \\ 1 \leq \; & j & \leq N. \end{aligned}$$

9.1 Partitioning

One partitions the convolution into N/R subproblems by partitioning the sum in Equation 9.1 as follows:

$$b_k \;=\; \sum_{l=1}^{N/R} \sum_{i+j=k+1} a_i \cdot w_j \tag{9.2}$$

where

$$\begin{aligned} 1 \leq \; & k \; \leq M + R - 1 \\ 1 \leq \; & i \; \leq M \\ (l-1)R + 1 \leq \; & j \; \leq lR + 1. \end{aligned}$$

This partitioning expresses the convolution of a and w as the sum of convolutions of a with N/R weight vectors w_j. Intuitively, one partitions weight vector w into chunks of length R, computes the partial convolutions, and exploits the associativity of the addition to form the sum of the partial convolutions when convenient.

9.2 Decoupling

Figure 9-1 demonstrates the data flow of a systolic convolution with $N = R$, while Figure 9-2 shows the pseudo code executed by each compute processor. This design is independent of the length M of vector a. The example in Figure 9-1 shows the case where $N = R = 4$ and $M = 5$. Both vector a and weight vector w enter the array from the left, and output vector b leaves the array on the right. Compute processor p_i is responsible for storing element w_i of the weight vector. Thus, the stream of elements w_i folds over on the way from left to right through the array. As elements of vector a are not stored for more than a single time step, they stream from left to right without folding over. During each time step, the compute processors multiply their local value w_i with the element of a_j arriving from the left, add the product to an intermediate value of b_k that is also received from the left, and send the new intermediate value to the right. The elements of b leave the array on the right.

The data movement in Figure 9-1 is illustrated by discussing the computation of $b_4 = a_4w_1 + a_3w_2 + a_2w_3 + a_1w_4$. The computation begins with time step 5 in Figure 9-1 when element a_4 enters processor p_1 on the left. Element w_1 is already resident. Processor p_1 computes the intermediate value $b_4^1 = a_4 \cdot w_1$, and sends it to processor p_2. At time step 6, p_2 receives a_3 and b_4^1 from processor p_1 on the left. With

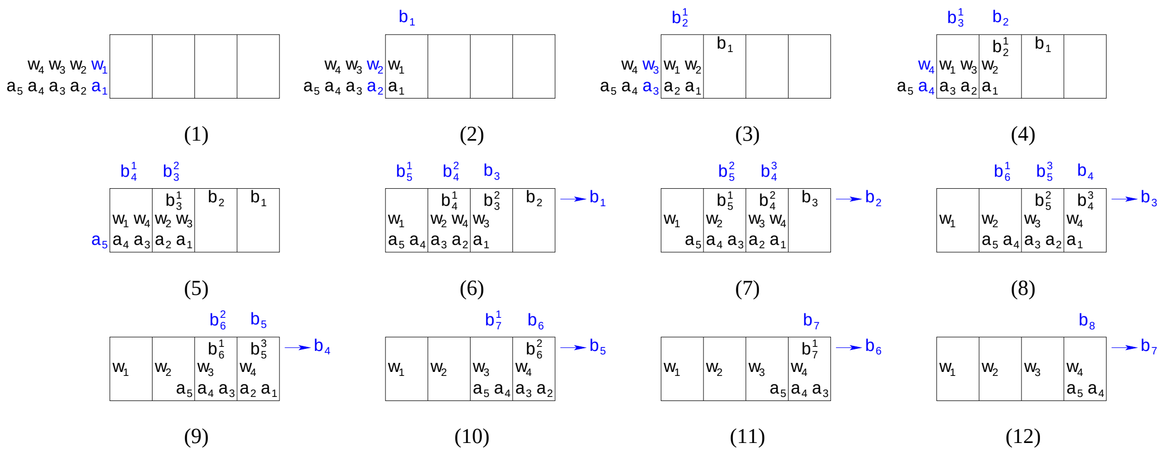


Figure 9-1: Systolic convolution of input sequence a_i of length $M = 5$ with $N = 4$ weights w_j. Both the weights and input sequence are fed into the linear array of $R = 4$ compute processors. Intermediate results are shown above the corresponding processors. Value b_k^i represents an intermediate value of b_k after the first i products have been computed according to Equation 9.1.

weight w_2 already resident, processor p_2 computes intermediate value $b_4^2 = b_4^1 + a_3 \cdot w_2$. In time step 7, values b_4^2, a_2, and w_3 are available for use by processor p_3. It computes and sends intermediate value $b_4^3 = b_4^2 + a_2 \cdot w_3$ towards processor p_4. At time step 8, p_4 receives b_4^3, a_1, and w_4 from p_3, and computes $b_4 = b_4^3 + a_1 \cdot w_4$. At time step 9, b_4 exits the compute array.

One uses the partitioning of Equation 9.2 to reduce a convolution with a weight vector of length N into N/R systolic convolutions that match network size R of a linear array of compute processors. In addition, one must employ one memory processor on the left of the array to buffer vectors a and w, and another memory processor on the right of the array to store intermediate values of the computation as well as to compute the sum of the subproblems. Figure 9-3 illustrates the computation of a convolution on a linear processor array. The decoupled systolic convolution requires $P = R$ compute processors and $M = 2$ memory processors. One may observe that $M = o(P)$ and, therefore, the convolution is decoupling efficient.

```
if $i = 1$
    $x_1 \leftarrow 0$
    $h \leftarrow$Net($west$)
    $a \leftarrow$Net($west$)$\times$Net($west3$) route Net($west3$) $\rightarrow x_2$
    for $n$: 1 .. $N-1$
        $a \leftarrow h\times$Net($west3$) route Net($west$) $\rightarrow$Net($east$),
                                          $a \rightarrow$Net($east2$),
                                          $x_1 \rightarrow$Net($east3$),
                                          Net($west3$) $\rightarrow x_2$,
                                          $x_2 \rightarrow x_1$
    for $m$: $n$ .. $M+i-1$
        $a \leftarrow h\times$Net($west3$) route $a \rightarrow$Net($east2$),
                                          $x_1 \rightarrow$Net($east3$),
                                          Net($west3$) $\rightarrow x_2$,
                                          $x_2 \rightarrow x_1$
    nop route $a \rightarrow$Net($east2$), $x_1 \rightarrow$Net($east3$), $x_2 \rightarrow x_1$
    nop route 0 $\rightarrow$Net($east2$), $x_1 \rightarrow$Net($east3$)

else if $i = R$
    $h \leftarrow$Net($west$)
    Net($east2$) $\leftarrow h\times$Net($west3$)+Net($west2$),
        route Net($west$) $\rightarrow h$
    for $m$: 2 .. $M+N-1$
        Net($east2$) $\leftarrow h\times$Net($west3$)+Net($west2$)

else
    $x_1 \leftarrow 0$
    $h \leftarrow$Net($west$)
    $a \leftarrow$Net($west$) $\times$ $h$+Net($west2$) route Net($west3$) $\rightarrow x_2$
    for $n$: 1 .. $N-i$
        $a \leftarrow h\times$Net($west3$)+Net($west2$) route Net($west$) $\rightarrow$Net($east$),
                                                       $a \rightarrow$Net($east2$),
                                                       $x_1 \rightarrow$Net($east3$),
                                                       Net($west3$) $\rightarrow x_2$,
                                                       $x_2 \rightarrow x_1$
    for $m$: n .. $M+N-1$
        $a \leftarrow h\times$Net($west3$)+Net($west2$) route $a \rightarrow$Net($east2$),
                                                       $x_1 \rightarrow$Net($east3$),
                                                       Net($west3$) $\rightarrow x_2$,
                                                       $x_2 \rightarrow x_1$
    nop route $a \rightarrow$Net($east2$), $x_1 \rightarrow$Net($east3$), $x_2 \rightarrow x_1$
    nop route 0 $\rightarrow$Net($east2$), $x_1 \rightarrow$Net($east3$)
```

Figure 9-2: Pseudo code for compute processor p_i in systolic convolution.

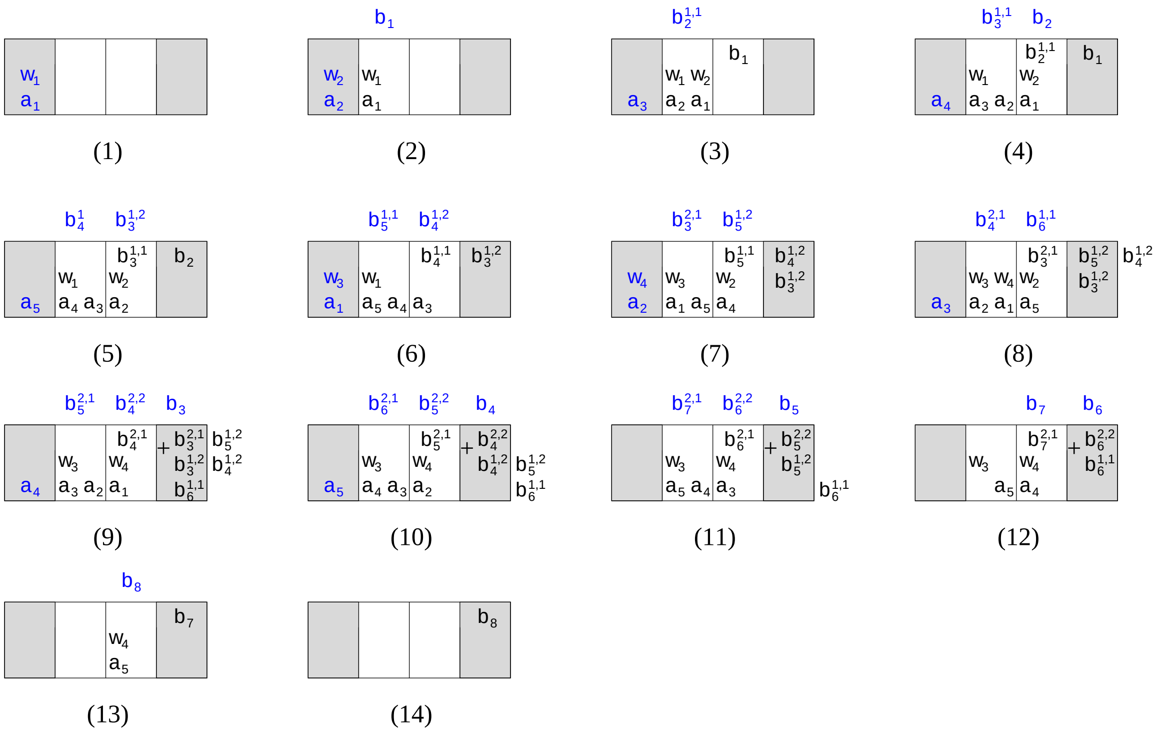


Figure 9-3: Stream convolution of an input sequence of length $M = 5$ with $N = 4$ weights on a linear array of $R = N/2 = 2$ compute processors and $M = 2$ memory processors. Value $b_k^{l,i}$ represents the computation of b_k when the outer summation of Equation 9.2 has been executed l times and the inner summation has been executed i times. Note that the memory tile on the right performs an addition to accumulate the results of the partial convolutions.

9.3 Efficiency Analysis

The number of multiply-and-add operations in the convolution of a sequence a of length M with a weight sequence w of length N is $C(M, N) = MN$. On a linear network of size R with $P(R) = R$ compute processors and $M = 2$ memory processors, one partitions the computation into $\sigma = N/R$ subproblems, each of which is a convolution of sequence a of length M with weight sequence w of length R. These subproblems overlap perfectly, as is obvious from Figure 9-3.

One calculates the number of time steps used by the convolution as follows[1]. There are $\sigma = N/R$ systolic convolutions on a linear array of R compute processors that

[1]The calculation for the efficiency of the convolution presented here corrects the earlier version presented in [15]. The previous version would be correct if the output vector b was to have the same length as the input vector a.

pipeline perfectly. Each of the systolic convolutions requires $M + R$ time steps to stream a sequence of length M through an array of size R, because each processor executes one multiply-and-add operation per time step. There are R time steps needed to drain the pipeline, but due to the perfect overlap of subsequent systolic convolutions, this penalty is incurred only once. Thus, the total number of time steps is:

$$T_{conv}(\sigma, R) = \sigma(M + R) + R.$$

Using a linear network of size R consisting of $P = R$ compute processors and $M = 2$ memory processors, the floating-point efficiency of the convolution is:

$$E_{conv}(\sigma, R) \quad = \quad \frac{\sigma}{\sigma + (N + R)/M} \cdot \frac{R}{R + 2} \tag{9.3}$$

Given the assumption that $M \geq N$, one has $N/M \leq 1$, and the efficiency of the stream-structured convolution approaches the optimal value of 100 % for large values of σ and R. Thus, for $\sigma \gg 1$, $E_{conv} \approx R/(R + 2)$ and one obtains more than 90 % efficiency for $R \geq 18$. For $N = R$ or, equivalently, $\sigma = 1$, the stream convolution reduces to a systolic convolution with a compute efficiency of

$$E_{conv}(\sigma = 1, R) \quad = \quad \frac{1}{1 + (N + R)/M} \cdot \frac{R}{R + 2}.$$

Chapter 10

Implementation on Raw

The Raw microprocessor [34, 33, 35] implements a superset of the features required for an architecture to be considered a decoupled systolic architecture[1], and thus Raw is an ideal platform for experimenting with stream algorithms. Implementing stream algorithms on Raw provides important insight into the behavior of stream algorithms and architectural issues that arise in practice. This chapter discusses a Raw implementation of each of the previous examples and compares these results with the best results one could obtain from an ***ideal DSA***.

The Raw architecture consists of sixteen identical, programmable ***tiles***. Each tile contains an 8-stage in-order single issue MIPS-like compute processor, a 4-stage pipelined FPU (for addition, subtraction, and multiplication), an unpipelined floating point divider, a 32kB data cache, a static communication router[2] (also called a switch or switch processor), and 96kB of instruction cache. The tiles are arranged in a four by four mesh network, and connected by two 32 bit wide static networks. The static router allows each tile to communicate with neighboring tiles to the north, south, east, and west over both static networks. Each tile is independently programmed, and within each tile, the instruction streams of the compute processor and the router are separated. Therefore, the router has a separate set of registers and a separate

[1]That Raw is a DSA should not be surprising given that the salient features of the DSA's interprocessor network are based on Raw's interconnect.

[2]Each tile contains an additional dynamic routers, but this feature is not of concern when treating Raw as a DSA.

program counter.

The introduction to this thesis compared a stream algorithm to a distributed memory implementation of a matrix multiplication on Raw. These results showed that the stream algorithm implementation ran much faster than the distributed memory implementation. The distributed memory matrix multiplication was implemented by assuming that each Raw tile computes one sixteenth of the result matrix, and that appropriate values of the operand matrices were replicated on the correct tiles. Such an implementation requires no communication (other than cache misses). However, implementation of any of the other examples in the distributed memory style would require communication. This creates an issue of credibility as the author believes that the most efficient form of communication for these problems on an architecture such as Raw is to structure them as stream algorithms. Therefore, while the comparison between stream algorithms and distributed memory implementation for the matrix multiply serves as good motivation to explore the idea of stream algorithms, this comparison will not be made for other examples.

However, in order to evaluate Raw as a DSA, a basis of comparison is still useful, and the ideal DSA serves this purpose. An ideal DSA does not suffer from loop overhead, branch mispredict penalties, or instruction cache misses. Furthermore, all functional units in an ideal DSA have a single cycle latency. Therefore, the only concern in implementing stream algorithms on an ideal DSA is scheduling the possibly heterogeneous computations of the stream algorithm in an efficient manner[3]. The efficiency analyses in previous sections assume an ideal DSA. By comparing the performance of stream algorithms implemented on Raw to that predicted for an ideal DSA, one gains insight into the degree to which practical architectural concerns, such as functional unit latency, effect efficiency and performance.

In order to interpret the results from Raw, it is important to understand how Raw differs from an ideal DSA and how those differences can effect performance:

Raw lacks a floating point multiply-and-add unit: Raw's lack of such a func-

[3]The concern of minimizing the overhead due to memory operations is of course accounted for by the use of stream algorithms in the first place.

tional unit means that one should expect the cycle counts of stream algorithms implemented on Raw to be twice that of those implemented on an ideal DSA. However, this structural concern should not effect the efficiency of stream algorithms implemented on Raw. When calculating efficiencies for Raw one simply doubles the total number of operations that need to be executed.

Raw has separate switch and compute processors: While each DSA processor has a single instruction stream that incorporates both routing and other instructions, each Raw tile has a compute processor and a switch processor. Thus, one must splice the instruction stream of a DSA processor into two separate instruction streams for execution on Raw. When translating DSA instructions to Raw instructions, one uses the route keyword to factor instruction streams. Operations on the left side of the **route** will be executed on Raw's compute processor, while the move operations on the right must be translated into router instructions. This creates additional difficulties for the programmer as all Raw programs must synchronize the switch and compute processors. While programming this synchronization can be difficult, it contributes only minimal overhead and thus is not a barrier to performance.

Raw's functional units have multi-cycle latency: Raw's floating point adder and multiplier both have four cycle latencies. Thus, simply translating a DSA's `fma` instruction into a multiply instruction followed by an add instruction does not result in an efficient Raw program, as the add instruction will stall waiting for the multiplication to finish. Again, this concern can be overcome by careful programming that translates groups of four `fma` instructions into four pairs of Raw multiplication and addition instructions and then schedules the multiplications before the additions. For example, this sequence of DSA instructions:

```
fma a, a, b, c
fma d, d, e, f
fma g, g, h, i
fma j, j, k, l
```

could be efficiently translated into the following sequence of Raw instructions:

```
mul b, b, c
mul e, e, f
mul h, h, i
mul k, k, l

add a, a, b
add d, d, e
add g, g, h
add j, j, k
```

However, this style of programming places additional burden on the programmer, and when the instructions operate on network data, it further complicates the communication between the switch processors and the compute processors. Furthermore, the data dependencies between `fma` instructions vary for different stream algorithms. Thus, the translation of a set of `fma` instruction into pairs of Raw multiplication and addition instructions is different for different algorithms. The correct translation for each algorithm is discussed below in the section on that algorithm.

Raw does not support the DSA memory interface: Each Raw compute processor is a single-issue pipeline designed to be programmed in a standard load/store manner and thus can execute only one memory operation (either a load or a store) per cycle. Thus, using a Raw tile to simulate a DSA's memory processor does not provide an adequate rate of memory accesses. However, the Raw simulator [32] is extensible and can be easily modified to meet the DSA's requirements. For these experiments, memory tiles are simulated as off-chip devices that can transfer one word per cycle from DRAM to Raw's network or from the network to DRAM. Additionally, the simulated memory tiles can perform simple reduction operations during the store transfer. A typical example from a stream algorithm is the operation `Mem[A] = Mem[A] - x`, where `A` is an address and `x` represents data received on the network.

Raw's divide unit has a twelve cycle latency and is not pipelined: The floating point divider on each Raw processor has a twelve cycle latency and is not pipelined. The decision to use a high latency and low occupancy divider in Raw was made by observing the fact that division typically occurs less frequently than multiplication and addition operations. This observation is consistent with the efficiency analyses done for each of the five example algorithms. However, the performance of stream algorithms that require divisions is effected by the latency of the divider for small data sizes. Furthermore, because stream algorithms reduce the instruction count by eliminating loads and stores from the critical path, the performance degradation due to a high latency division is more pronounced for stream algorithms than for more traditional methods of programming[4]. The effects of Raw's divider on stream algorithm performance are discussed in more detail in the following discussion of individual examples.

Raw's input ports are not symmetric: I/O is a first class citizen on Raw, and Raw's pins are an extension of its static networks. However, due to packaging constraints some of the outputs from the static networks are multiplexed onto a single pin. This sharing of resources means that some networks can receive only one word every other cycle instead of one word per cycle. Careful scheduling of instructions can alleviate this concern, but again it places an additional burden on the programmer.

Keeping these performance issues in mind, all five stream algorithms have been implemented using the Raw simulator. The simulator was configured to support a single Raw chip consisting of a four by four array of tiles. Each tile in this array was programmed as a compute tile, and thus only operated on data in its local registers or from the network. Memory tiles were simulated using off chip devices that could

[4]A similar effect has been observed while increasing the issue width of super-scalar machines [29]. As more instruction level parallelism is exploited the latency of the divider is exposed to a greater degree, and has a greater effect on performance.

deliver the required rate of memory operations. For each algorithm, the performance was tested on a number of data sizes and compared to that predicted for an ideal DSA. The implementation and results of each individual algorithm are discussed below.

Each algorithm is implemented in a fully parameterized manner, so that the problems size, represented by M and N, could easily be modified. This parameterization allows one to easily experiment with different problem sizes, and thus different values of σ, where σ is the ratio of the problem size to the network size, R. For the Raw implementations of stream algorithms, the network size was fixed at $R = 4$, which is the standard configuration for the prototype Raw processor. This parameterized method of implementation is very flexible, but suffers from small overhead cost that can effect the efficiency of small problem sizes.

For each algorithm, both the performance of the compute processors and the overall system performance (including the simulated memory processors) are presented as a function of the problem size and σ, respectively. The performance of the compute tiles provides insight into how stream algorithms could benefit the current Raw system, and how stream algorithms could perform for large computational fabrics. The performance of the system, including compute tiles, provides insight into the efficiency of Raw as a DSA, and it provides a direct comparison for the efficiency analysis in previous sections.

10.1 Matrix Multiplication

The matrix multiplication is the first example of a stream algorithm on Raw. The implementation is a fairly straightforward mapping of the DSA pseudo code from Figure 5-2 to Raw code.

Because the matrix multiplication is so simple, it serves as a good baseline for evaluating Raw's performance when executing stream algorithms. Because there are no divisions and there is abundant instruction level parallelism, analyzing the performance on small data sizes can give insight into the overhead of this approach.

The amount of instruction level parallelism and thus the performance of the matrix multiplication can be increased by adding three extra additions to the computation performed on each processor. On the DSA, each processor performs an inner product:

```
c = 0.0;
for(i = 0; i < N; i++)
  fma c, c, $W1, $N2
```

As described above, this loop should be translated into the following efficient Raw code:

```
c = 0.0;
for(i = 0; i < N; i+=4) {
  mul temp1, $W1, $N2
  mul temp2, $W1, $N2
  mul temp3, $W1, $N2
  mul temp4, $W1, $N2

  add c, c, temp1
  add temp2, temp2, temp3
  add c, c, temp2
  add c, c, temp4
}
```

However, due to the dependencies among the add instructions and the latency of the floating point unit this code will be plagued by bypass stalls when executed. Furthermore, these stalls will occur each time through the loop. Therefore, this loop is not satisfactory. However, one may eliminate all of these stalls by adding a small amount of work to be done after the completion of the loop:

```
c = 0.0;
c1 = 0.0;
c2 = 0.0;
c3 = 0.0;
```

```
c4 = 0.0;
for(i = 0; i < N; i+=4) {
  mul temp1, $W1, $N2
  mul temp2, $W1, $N2
  mul temp3, $W1, $N2
  mul temp4, $W1, $N2

  add c1, c1, temp1
  add c2, c2, temp2
  add c3, c3, temp3
  add c4, c4, temp4
}
add c1, c1, c2
add c3, c3, c4
add c, c1, c3
```

This code executes without any bypass stalls during the loop. For large values of N, the additional, post-loop `add`'s and their associated bypass stalls have a minimal effect on performance.

By carefully considering such performance issues, one is able to execute a highly efficient matrix multiplication on Raw. Figure 10-1 shows the efficiency of the compute processors executing the matrix multiplication as a function of the problem size, N, while Figure 10-2 shows the efficiency of the entire system (including memory processors) as a function of σ. For comparison, the results predicted for an ideal DSA are also shown. Both x-axes use a logarithmic scale.

As these results indicate, Raw can achieve very high performance when executing stream algorithms, even for small data sizes. In this case the data sizes range from $N = 16$ to $N = 1024$, and the efficiency of the compute processors range from 55 % to 96 %. Even for small problem sizes these efficiencies are very high for a general purpose processor. These results represent the benefits of the stream-structuring methodology.

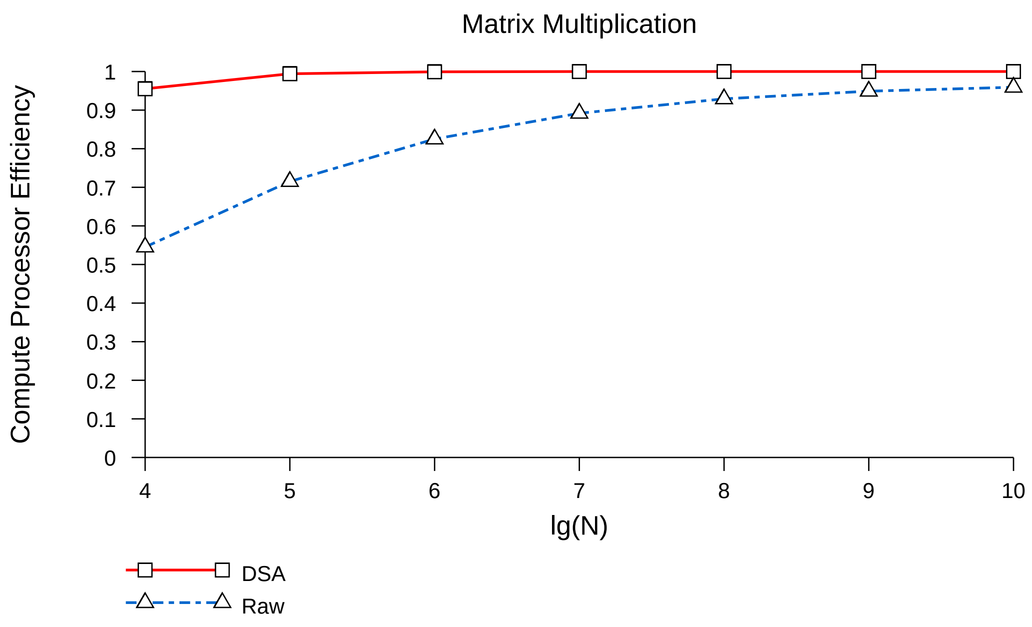


Figure 10-1: The efficiency of the compute processors executing a stream-structured matrix multiplication is shown as a function of N. The solid curve represents the results predicted by the efficiency analysis for an ideal DSA, while dotted curve represents measured results on Raw.

The difference between the performance on Raw and the performance of an ideal DSA at small data sizes is due almost entirely to the manner in which the stream algorithm was implemented. The matrix multiplication was done in a parameterized fashion that allowed the data sizes to be easily changed. This design decision means that a number of conditional statements must be executed in order to handle a number of situations. For small data sizes, these conditional instructions and any mispredictions can have a large effect on performance. If the algorithm had been optimized for each particular data size, one would expect to see Raw's performance closer the predicted performance for an ideal DSA.

10.2 Triangular Solver

The second stream algorithm implemented on Raw is the triangular solver. When partitioned, the triangular solver consists of two distinct systolic algorithms, one that implements a triangular solver and one that updates intermediate values via matrix

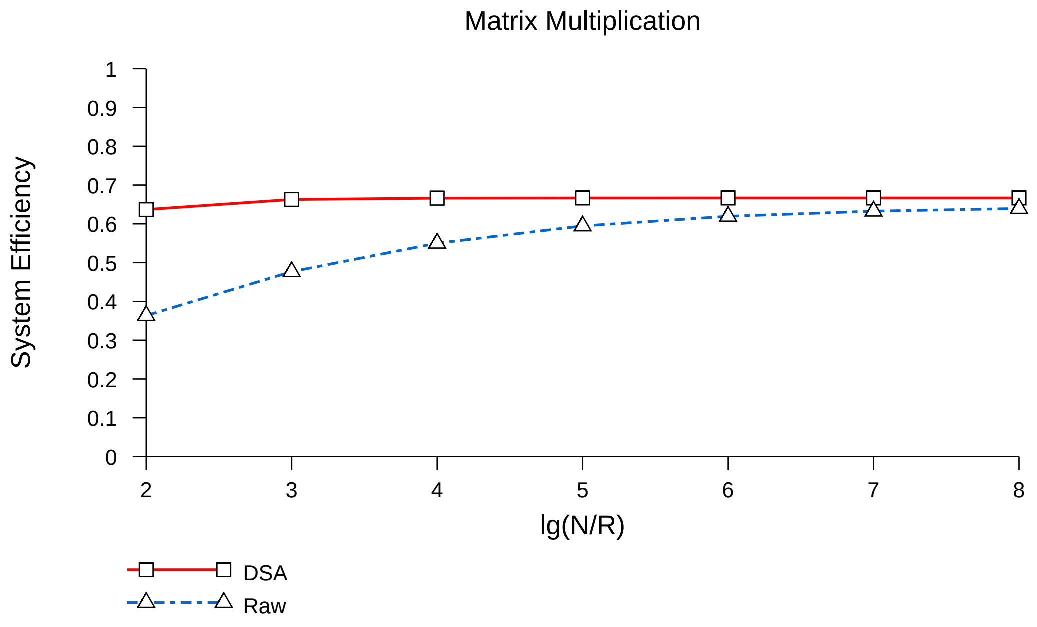


Figure 10-2: The efficiency of all processors (both memory and compute) executing a stream-structured matrix multiplication is shown as a function of $\sigma = N/R$, where $R = 4$. The solid curve represents the results predicted by the efficiency analysis for an ideal DSA, while the dotted curve represents physical results measured on Raw.

multiplication.

The matrix multiplication used here differs slightly from the matrix multiplication described in the preceding section. The preceding systolic matrix multiplication was not optimized for small problem sizes. However, the update operations in the triangular solver consist of a series of square $R \times R$ matrix multiplications. For these experiments, R was fixed such that $R = 4$. Therefore, the matrix multiplication used for the triangular solver was optimized to handle pipelined square 4×4 matrix multiplications.

As noted in the section on optimizing the matrix multiplication for Raw, the data dependencies in the inner product computation can severely limit performance. This is also the case when optimizing Raw code for the pipelined 4×4 problem size required by the triangular solver. However, in this case, one cannot extract more parallelism by unrolling the loop, simply because the loop is already so short. In this case, one may make each processor responsible for four times the number of output values. Overlapping the computation of four separate output values per processor provides enough instruction level parallelism to schedule the code efficiently on Raw. For values of N which are multiples of 4 and greater than 16, using this technique will result in very efficient code. If N does not meet these criteria, the code cannot be scheduled without bypass stalls and will not be maximally efficient. However, for large problem sizes, the number of bypass stalls is insignificant.

Unlike the matrix multiplication, the performance of the systolic array for the triangular solver is limited by the latency of Raw's floating point divider. The divider has a twelve cycle latency, and its result is sent to a special purpose register, which takes an additional instruction to access. Furthermore, there is an interprocessor dependence on the results of the divide operations. That is, the divide operations performed by processors in row i must wait for the divide operations of the processors in row $i-1$ to complete before executing. For small problems sizes, the latency of the floating point divider has a profound effect on performance.

The results shown in Figure 10-3 and Figure 10-4 confirm that for small problem sizes, the triangular solver executed on Raw has much lower performance than what one would expect for an ideal DSA. Figure 10-3 shows the efficiency of the compute processors as a function of the problem size, N, while Figure 10-4 shows the efficiency of the overall system (including memory processors) as a function of σ.

One should also compare the performance of the lower triangular solver to that of the matrix multiplication. While both algorithms are very efficient for large problem sizes, the performance of the triangular solver on small problem sizes is much lower that that of the matrix multiplication. This difference in performance has two main components. The first is that for a given value of N, the triangular solver does half as much work as the matrix multiplication. This means that overhead costs are more pronounced for the triangular solve than for the matrix multiplication, as there are less useful operations to amortize the startup costs. The other component to the performance of the triangular solver is Raw's floating point divider. For small problems sizes, where the number of divides is significant, these results suggest that Raw would benefit from a lower-latency divider.

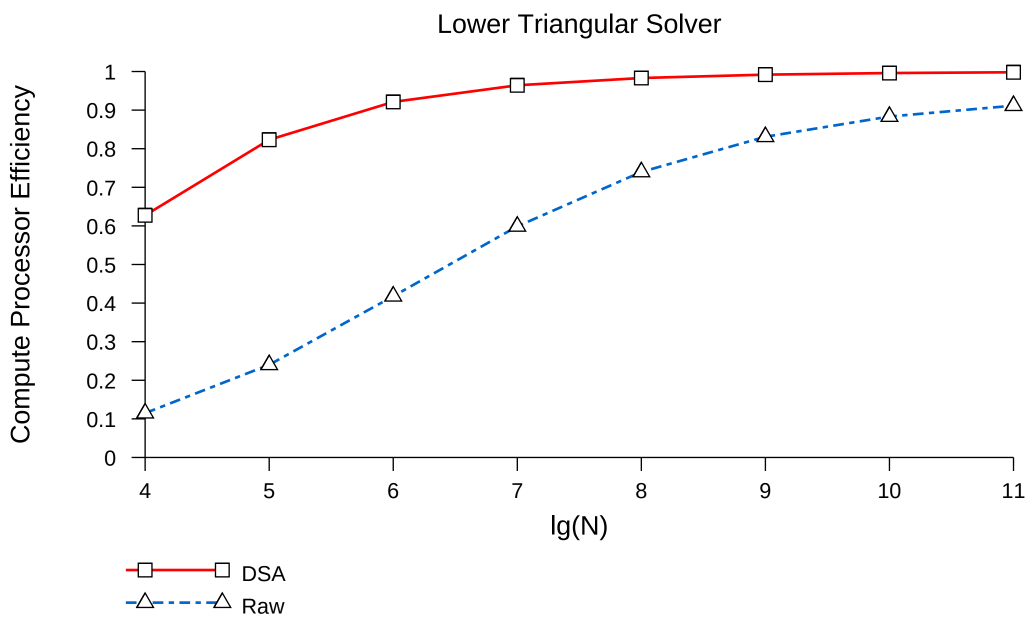


Figure 10-3: The efficiency of the compute processors executing a stream-structured lower triangular solver is shown as a function of N. The solid curve represents the results predicted by the efficiency analysis for an ideal DSA, while the dotted curve represents physical results using Raw to implement a DSA.

10.3 LU Factorization

The LU factorization is the third example of implementing a stream algorithm on Raw. The partitioning for this problem requires four systolic subproblems: an LU factorization, a lower triangular solver, an upper triangular solver, and a matrix multiplication. Like the triangular solvers, the performance of the systolic algorithm for LU factorization operating on small problem sizes is limited by the latency of Raw's floating point divider and by overhead costs.

The performance issues in the LU factorization are very similar to those of the triangular solver. Like the triangular solver, the systolic algorithm for LU factorization has similar interprocessor data dependencies, which expose the latency of Raw's floating point divider. Furthermore, the updates to intermediate values of the matrix multiplication are effected by the same issues that effect the systolic matrix multiplication for the triangular solver, and the same optimizations are used in this case.

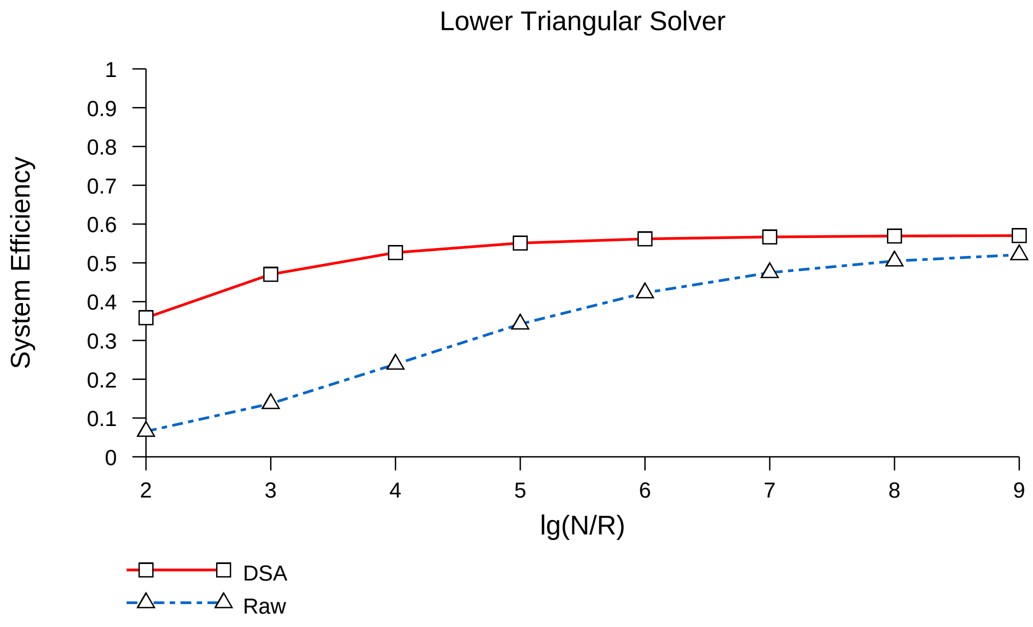


Figure 10-4: The efficiency of all processors executing a stream-structured lower triangular solver is shown as a function of $\sigma = N/R$, where $R = 4$. The solid curve represents the results predicted by the efficiency analysis for an ideal DSA, while the dotted curve represents physical results measured on Raw.

Figure 10-5 shows the efficiency of the compute processors executing an LU factorization as a function of N, while Figure 10-6 shows the overall system performance as a function of $\sigma = N/R$. These results are very similar to the results for the triangular solver. For small data sizes, the performance of the divider and the overhead costs keep performance on Raw low compared to that of the ideal DSA. However, as the problem size increases, the efficiencies get very high.

10.4 QR Factorization

The QR factorization is the fourth, and most complicated, example of implementing a stream algorithm on Raw. All results assume that the matrix being factored is an $N \times N$ matrix. Figure 10-7 shows the efficiency the compute processors computing matrix R as a function of the problem size N. Figure 10-8 shows the efficiency of the entire system (including memory processors) computing matrix R as a function of σ. Figure 10-9 shows the efficiency of computing the entire QR factorization as a

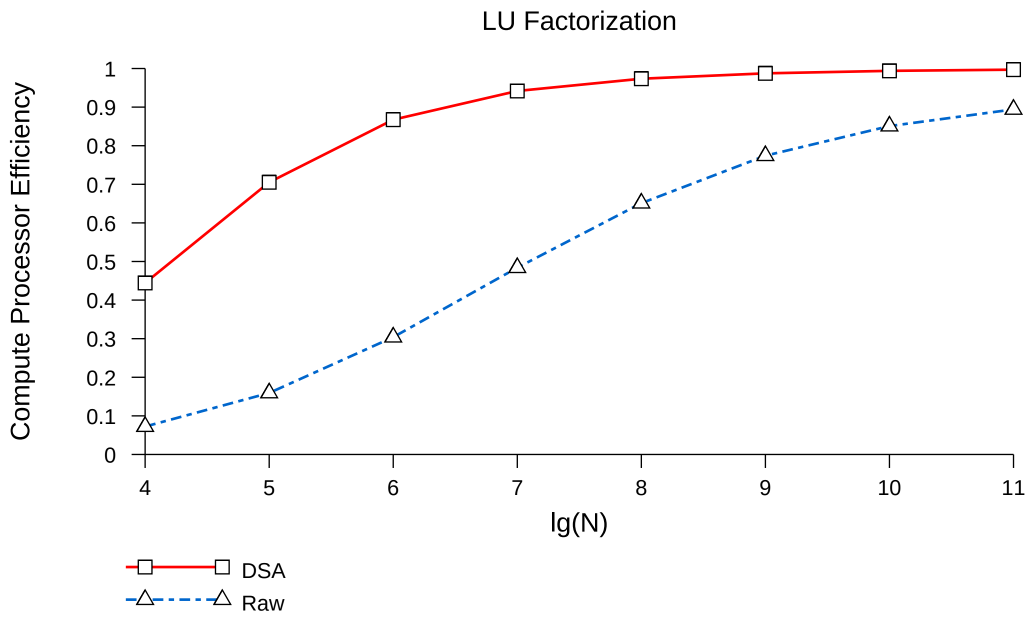


Figure 10-5: The efficiency of the compute processors executing a stream-structured LU factorization is shown as a function of N. The solid curve represents the results predicted by the efficiency analysis, while the dotted curve represents physical results measured on Raw.

function of the problem size N. Figure 10-10 shows the efficiency of all processors computing the QR factorization as a function of σ.

There are two issues that arise in translating the pseudocode of Figures 8-4 and 8-9 into efficient Raw code. The first is the fact that the pseudocode makes use of the DSA's ability to simultaneously operate on network port and route the data from that port into a register, as shown in this line of pseudocode from Figure 8-9:

```
$\text{Net}(south) \leftarrow x \times \text{Net}(east) + \text{Net}(north)$
    route $\text{Net}(east) \rightarrow \text{Net}(west)$, $\text{Net}(north) \rightarrow a$
```

The compute processor on a Raw tile cannot simultaneously route data from the network into a functional unit and into a register. However, the static router on a Raw tile can route data to the compute processor and into one of its own registers simultaneously. That data can then be sent from the router register to the compute processor as soon as the next cycle. Using this technique helps one to translate the DSA pseduocode into efficient Raw code.

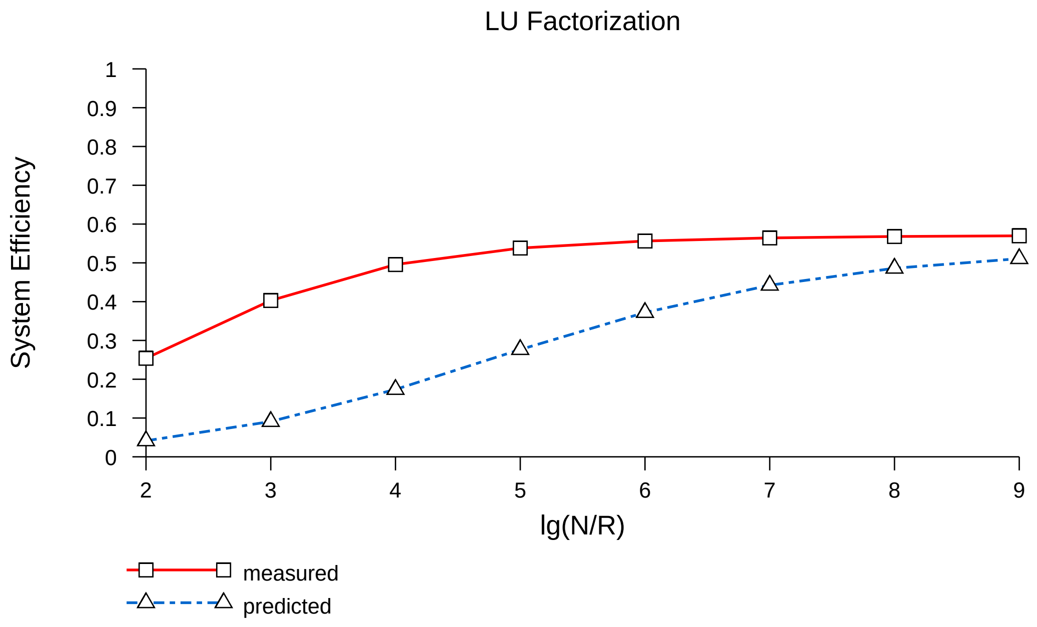


Figure 10-6: The efficiency of all processors executing a stream-structured LU factorization is shown as a function of $\sigma = N/R$, where $R = 4$. The solid curve represents the results predicted by the efficiency analysis for an ideal DSA, while the dotted curve represents physical results measured on Raw.

The second issue of which one must be aware when implementing a stream-structured QR factorization on Raw is the loop-carry dependence seen in the pseudocode of Figure 8-9. The value of x calculated in loop iteration n is dependent on the value calculated in iteration $n - 1$. This is not a problem for the DSA, with its single cycle `fma` latency. However, on Raw, with its multicycle functional unit latency, this dependence can become a performance barrier. However, one may increase the instruction level parallelism and thus avoid bypass stalls by computing multiple independent outputs on a single Raw tile. Computing four output elements per tile increases the instruction level parallelism to the point that the code executes without bypass stalls. For the systolic premultiplication, the columns are independent, and thus the computation of four separate columns can be grouped together. For systolic postmultiplication, the computation of separate rows may be grouped. For the systolic Givens computation, the time spent performing and waiting for division operations dominates the execution time and this optimization is insignificant.

The design of Raw's floating point divider has an even more significant effect on

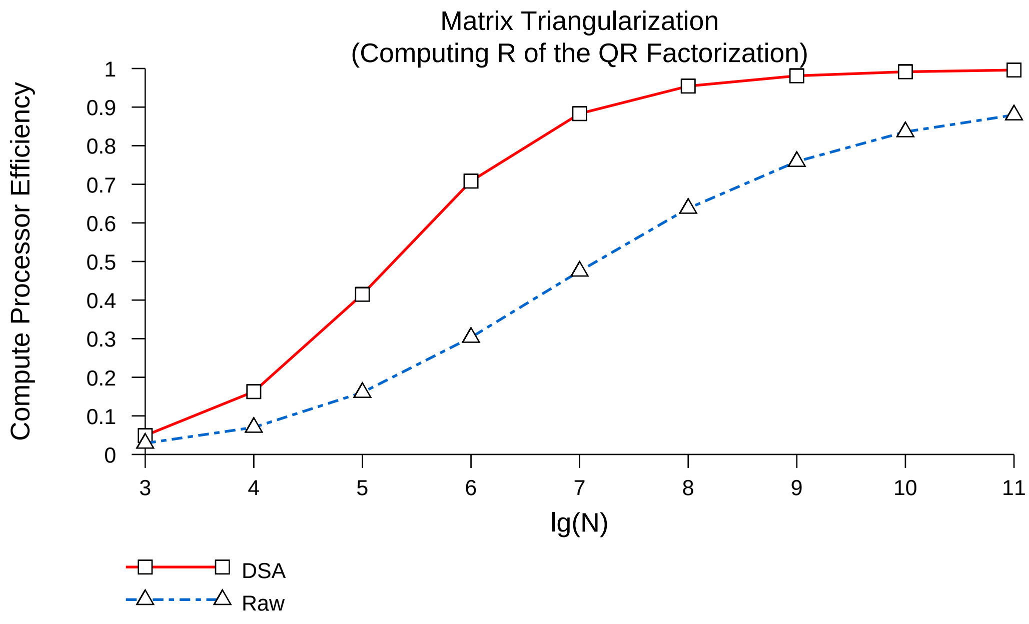


Figure 10-7: The efficiency of the compute processors executing a stream-structured QR factorization and computing only R is shown as a function of N. The solid curve represents the results predicted by the efficiency analysis for an ideal DSA, while the dotted curve represents physical results measured on Raw.

the performance of the QR factorization than in the LU factorization or the triangular solver. As one can see in Figure 8-4, the systolic Givens computation must execute one division to compute a value α, and then must immediately execute another division to compute a value β. All subsequent computations done by this tile depend on these values. Here both the twelve-cycle latency of Raw's divider and the fact that it is not pipelined severely limit the performance of the QR factorization. The systolic premultiplication and postmultiplication operations can be implemented extremely efficiently, but the size of the matrix must be very large before the efficiency of these operations dominates the total number of cycles. The difference between the Raw results and those of an ideal DSA is not a failure of stream algorithms. Rather it serves to highlight the point that when implementing algorithms that are designed to execute one floating point operation per cycle, the effects of a long latency operation, even one as seldom used as division, become readily apparent.

The latency of the square root operation also effects the performance of the QR factorization on Raw. On Raw, square roots are calculated in software and take

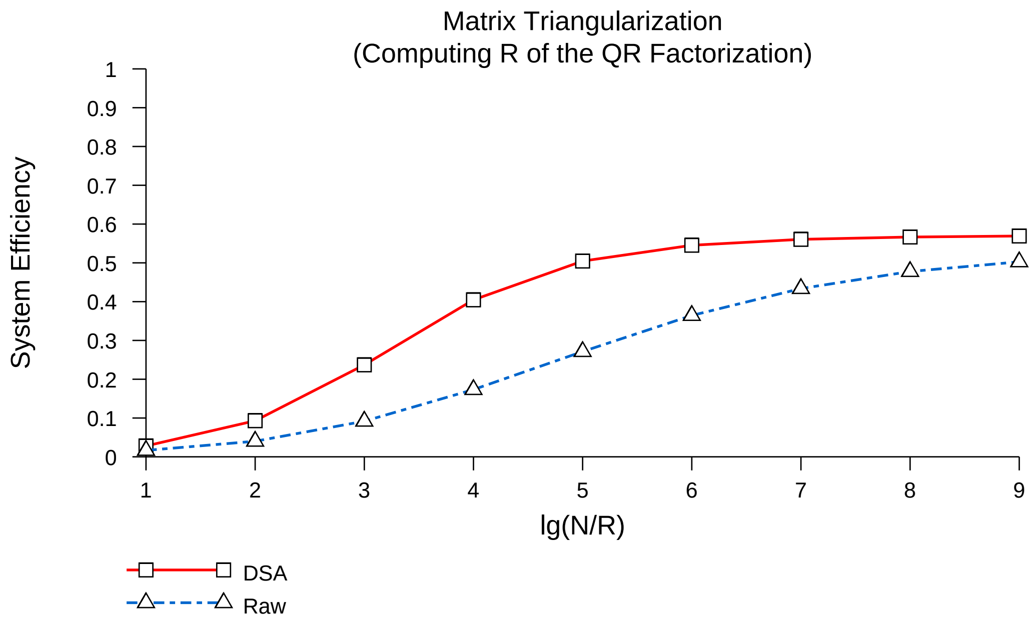


Figure 10-8: The efficiency of all processors executing a stream-structured QR factorization, but omitting the computation of Q is shown as a function of $\sigma = N/R$, where $R = 4$. The solid curve represents the results predicted by the efficiency analysis, while the dashed curve represents physical results using Raw to implement a DSA.

hundreds of cycles to complete. However, square roots are only computed on the diagonal of the processor array and each diagonal processor computes only a single square root operation per systolic Givens computation and there are no dependencies between square root operations. Therefore, the latency of this operation has a minimal effect on performance and then only for very small values of σ.

As mentioned in the efficiency analysis of the QR factorization in Chapter 8, there is a better algorithm for QR factorization in terms of numerical stability. This alternative form of fast Givens rotations involves using different types of rotations depending on the characteristics of the input data. In addition to computation, this algorithm involves conditional statements that evaluate the type of Givens transformations. Due to these extra instructions, the upper bound for the efficiency on the DSA executing a QR factorization based on this second type of givens transformation would be 66 %. However, on Raw the efficiency of this implementation would be much worse. In addition to the extra overhead of the branch instruction itself, the Raw implementation would suffer from branch mispredict penalties. Furthermore,

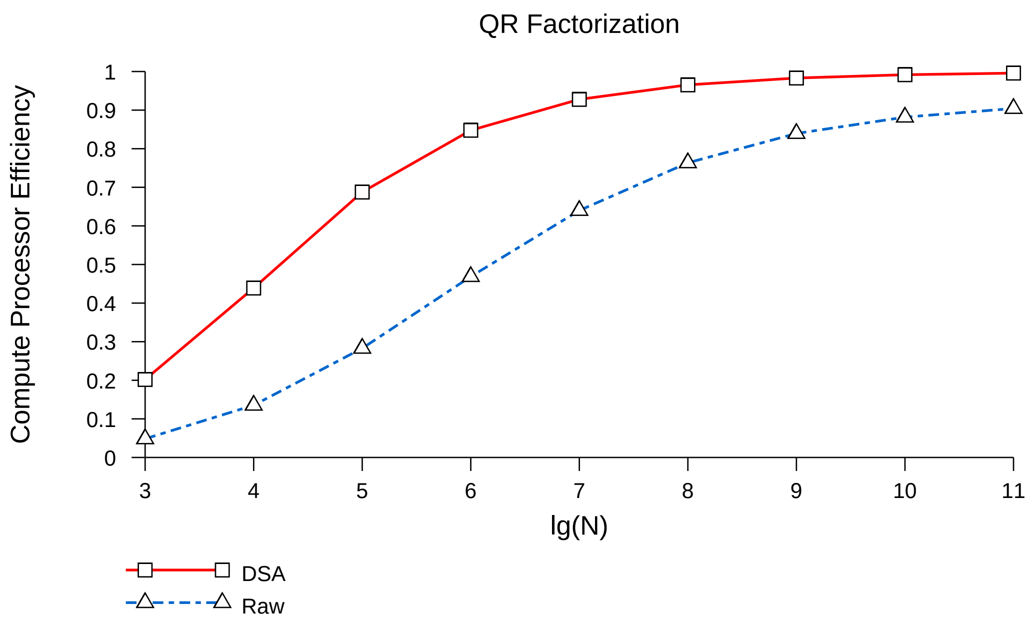


Figure 10-9: The efficiency of the compute processors executing a stream-structured QR factorization is shown as a function of N. The solid curve represents the results predicted by the efficiency analysis, while the dotted curve represents results measured on Raw.

having the conditional statements in the loop would make optimizing the instruction schedule to eliminate bypass stalls more difficult. It is possible that for large data sizes, slow Givens transformations may be faster than this alternate form of fast Givens transformations. Although slow Givens transformations require more square root operations and more `fma` operations, they require no conditional code (and no branch mispredicts) and thus might be easier to optimize for Raw.

10.5 Convolution

Convolution is the final stream algorithm implemented on Raw. As described in Chapter 9, convolution is an operation on two vectors, one of length M and one of length N, where it is assumed that $M \geq N$. Figure 10-11 shows the efficiency of the compute processors executing a stream-structured convolution as a function of N, while Figure 10-12 shows the efficiency of the system (including memory processors) as a function of $\sigma = N/R$. In this case, different curves show results obtained on Raw

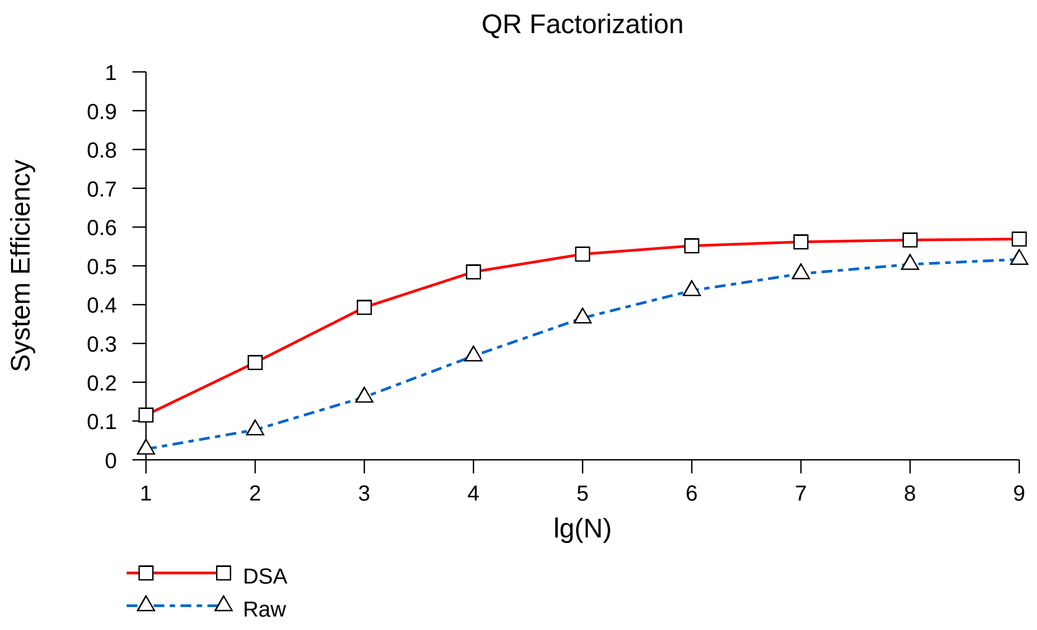


Figure 10-10: The efficiency of the all processors executing a stream-structured QR factorization is shown as a function of $\sigma = N/R$, where $R = 4$. The solid curve represents the results predicted by the efficiency analysis for an ideal DSA, while the dotted curve represents physical results measured on Raw.

for different values of M. These results are compared to the results predicted for an ideal DSA[5].

To translate the DSA pseudocode of Figure 9-2 to Raw code, one must account for the fact that Raw has only two networks connecting processors and not the three networks assumed by the DSA. This turns out not to be a problem because Raw also lacks a floating point multiply-and-add instruction. Thus a DSA `fma` instruction is translated into two Raw instructions, and Raw's two networks can easily deliver three operands in two cycles.

One should also note that, like the QR factorization, the DSA pseudocode for convolution uses instructions that simultaneously route data into a functional unit and a register. However, once again the programmer can use the switch processor's registers to temporarily store the value rather than routing it into Raw's compute processor.

[5]Only one curve is shown for the ideal DSA, because the predictions for all three sizes of M vary only slightly

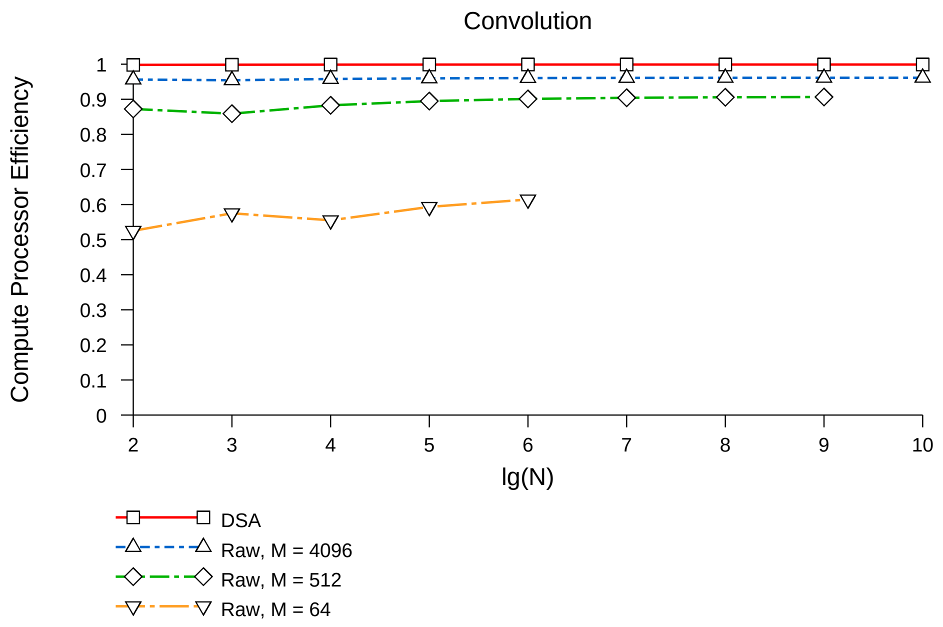


Figure 10-11: The efficiency of the compute processors executing a stream-structured convolution is shown as a function of the length N of the shorter vector. The solid curve represents the results predicted by the efficiency analysis for an ideal DSA, while the dotted curves represents physical results measured on Raw for three separate data sizes.

For a stream-structured convolution implemented on Raw, the performance depends on the size of M, and not the size of N. The loss of performance for small sizes of M is due entirely to the overhead associated with the parameterized approach taken to implement these examples. Like the other examples, the performance of the convolution for small values of M on Raw would benefit significantly from optimizing for particular data sizes.

10.6 Summary

Analyzing performance data for the above examples leads to the following observations:

- Stream algorithms achieve high efficiencies, not just in theory, but also in practice.
- Stream algorithms are an efficient way to program single chip tiled architectures

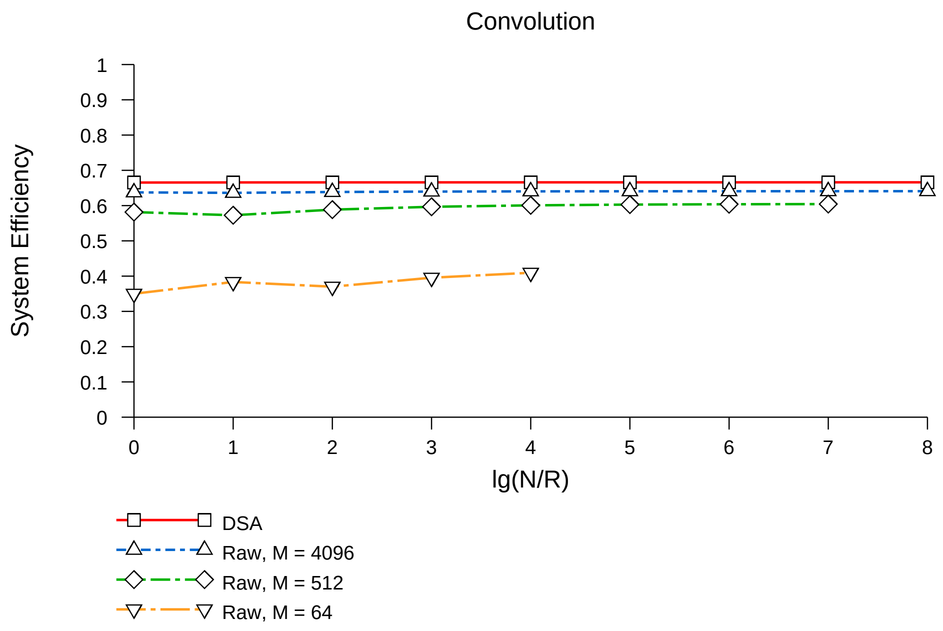


Figure 10-12: The efficiency of all processors executing a stream-structured convolution is shown as a function of $\sigma = N/R$, where $R = 4$ and N is the length of the shorter vector. The solid curve represents the results predicted by the efficiency analysis, while the dotted curves represents physical results for three different data sizes using Raw to implement a DSA.

such as Raw.

- For small data sizes better performance could be obtained by tailoring the implementation for that specific size.

- Stream algorithms would execute almost twice as fast on Raw if Raw had a floating point multiply-and-add unit.

- The performance of the floating point divider has a significant impact on performance of stream algorithms for small data sizes.

- The performance per area of Raw could be increased by eliminating Raw's data caches and adding more functional units.

- The system performance for all examples would improve if executed on a larger Raw fabric.

- The performance of all examples is expected to continue increasing as problem size increases[6].

[6]The amount of time required to run the Raw simulator for large problem sizes limited the amount of data that could be collected before publication of this thesis.

Chapter 11

Conclusion

This thesis has presented stream algorithms and the decoupled systolic architecture, which comprise a scalable means of performing highly efficient computation. Both the scalability and the efficiency of this approach are due to a design methodology that uses a bounded amount of storage for each processing element and reduces the amount of memory operations on the critical path of computation. The theoretical treatment of stream algorithms on an ideal DSA has shown that the efficiency of this approach can reach 100 % asymptotically, while the experimental treatment on Raw has shown that stream algorithms achieve high efficiencies in practice.

Unlike systolic arrays, the decoupled systolic architecture allows one to execute programmed stream algorithms of arbitrary problem size on a constant-sized machine. In contrast to contemporary parallel RISC architectures, the decoupled systolic architecture enables one to increase efficiency by increasing the number of processors.

This work presented five concrete examples of stream algorithms for a matrix multiplication, a triangular solver, an LU factorization, a QR factorization, and a convolution. All examples were analyzed theoretically and implemented on the Raw microprocessor. Table 11.1 summarizes the results for these applications. For each stream algorithm, the table lists the number of compute processors P and memory processors M as a function of network size R. Table 11.1 also compares the execution times T and efficiencies E of stream algorithms, and presents the maximum compute processor efficiency achieved on a 4×4 Raw configuration E_{raw}.

App.	$P(R)$	$M(R)$	$T(\sigma, R)$	$E(\sigma, R)$	$E_{raw}(\sigma)$
mat. mult.	R^2	$2R$	$\sigma^3 R + 3R$	$\frac{\sigma^3}{\sigma^3+3} \cdot \frac{R}{R+2}$	.96
tri. solver	R^2	$3R$	$\frac{R}{2}(\sigma^3 + \sigma^2 + 6\sigma - 2)$	$\frac{\sigma^3}{\sigma^3+\sigma^2+6\sigma-2} \cdot \frac{R}{R+3}$	.91
LU fact.	R^2	$3R$	$R(\frac{1}{3}\sigma^3 + \frac{1}{2}\sigma^2 + \frac{31}{6}\sigma - 2)$	$\frac{\sigma^3}{\sigma^3+\frac{3}{2}\sigma^2+\frac{31}{2}\sigma-6} \cdot \frac{R}{R+3}$	.89
QR fact.	R^2	$3R$	$R(\frac{5}{3}\sigma^3 + \frac{7}{2}\sigma^2 + \frac{125}{6}\sigma - 3)$	$\frac{\sigma^3}{\sigma^3+\frac{21}{10}\sigma^2+\frac{25}{2}\sigma-\frac{9}{5}} \cdot \frac{R}{R+3}$	.90
conv.	R	2	$\sigma(M + R) + R$	$\frac{\sigma}{\sigma+(N+R)/M} \cdot \frac{R}{R+2}$	.96

Table 11.1: Summary of stream algorithms. The table shows the number of compute processors P and the number of memory processors M. In addition, the table compares the execution time T and compute efficiency E and compares that to the maximum compute processor efficiency achieved on a 4×4 configuration of Raw tiles, E_{raw}.

The experience with the design of stream algorithms has revealed three noteworthy insights. First of all, the design philosophy for stream algorithms appears to be quite versatile. One is able to formulate stream algorithms even for relatively complex algorithms like the QR factorization. Secondly, stream algorithms achieve high compute efficiency on a general purpose architecture where conventional designs have not. Third, stream algorithms exploit short, fast wires making them a good match for modern architectures concerned with wire delay.

Bibliography

[1] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, 2nd edition, 1996.

[2] Marco Annaratone, Emmanuel Arnould, Thomas Gross, H. T. Kung, Monica S. Lam, Onat Menzilcioglu, Ken Sarocky, and Jon A. Webb. Warp Architecture and Implementation. In *13th Annual Symposium on Computer Architecture*, pages 346–356, 1986.

[3] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jeenifer Widom. Models and Issues in Data Stream Systems. In *Proceedings of 21st ACM Symposium on Principles of Database Systems*, 2002.

[4] Ziv Bar-Yossef, Omer Reingold, Ronen Shaltiel, and Luca Trevisan. Streaming Computation of Combinatorial Objects. In *Complexity*, 2002.

[5] Manuel E. Benitez and Jack W. Davidson. Code Generation for Streaming: an Access/Execute Mechanism. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 132–141, 1991.

[6] A. Bojanczk, R. P. Brent, and H. T. Kung. Numerically Stable Solution of Dense Systems of Linear Equations Using Mesh Connected Processors. Technical report, Carnegie-Mellon University, Department of Computer Science, May 1981.

[7] Shekhar Borkar, Robert Cohn, George Cox, Thomas Gross, H. T. Kung, Monica Lam, Margie Levine, Brian Moore, Wire Moore, Craig Peterson, Jim Susman,

Jim Sutton, John Urbanski, and Jon Webb. Supporting Systolic and Memory Communication in iWarp. In *17th International Symposium on Computer Architecture*, pages 70–81, Seattle, WA, May 1990.

[8] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, MA, 1990.

[9] Erik Elmroth and Fred G. Gustavson. Applying recursion to serial and parallel QR factorization leads to better performance. *IBM Journal of Research and Development*, 44(4):605–624, 2000.

[10] W. M. Gentleman and H. T. Kung. Matrix triangularization by systolic arrays. In *Proceedings of SPIE Symposium, Vol. 298, Real-Time Signal Processing IV, The Society of Photo-optical Instrumentation Engineers*, August 1981.

[11] Gene H. Golub and Charles F. Van Loan. *Matrix Computations.* John Hopkins University Press, Baltimore and London, 2nd edition, 1993.

[12] Thomas Gross and Monica S. Lam. Compilation for a high-performance systolic array. In *Proceedings of the 1986 SIGPLAN symposium on Compiler contruction*, pages 27–38. ACM Press, 1986.

[13] Daniel W. Hillis. *The Connection Machine.* MIT Press, Cambridge, MA, 1985.

[14] Ron Ho, Kenneth W. Mai, and Mark A. Horowitz. The Future of Wires. *Proceedings of the IEEE*, 89(4):490–504, April 2001.

[15] Henry Hoffmann, Volker Strumpen, and Anant Agarwal. Stream Algorithms and Architecture. Technical Report MIT-LCS-TM-636, MIT Laboratory for Computer Science, Cambridge, MA, March 2003.

[16] R.C. Holt and Tom West. *Turing Reference Manual (Seventh Edition).* Holt Software Associates Inc.

[17] Dror Irony and Sivan Toledo. Communication-efficient Parallel Dense LU Using a 3-dimensional Approach. In *Proceedings of the 10th SIAM Conference on Parallel Processing for Scientific Computing*, March 2001.

[18] Ujval J. Kapasi, Peter Mattson, William J. Dally, John D. Owens, and Brian Towles. Stream Scheduling. In *Proceedings of the 3rd Workshop on Media and Stream Processing*, Austin, TX, December 2001.

[19] Brucek Khailany, William J. Dally, Ujval J. Kapasi, Peter Mattson, Jinyung Namkoong, John D. Owens, Brian Towles, Andrew Chang, and Scott Rixner. Imagine: Media Processing with Streams. *IEEE Micro*, 21(2):35–46, March/April 2001.

[20] H. T. Kung. Why Systolic Architectures? *IEEE Computer*, 15(1):37–46, January 1982.

[21] H. T. Kung and Charles E. Leiserson. Algorithms for VLSI Processor Arrays. In Carver A. Mead and Lynn A. Conway, editors, *Introduction to VLSI Systems*, chapter 8.3, pages 271–292. Addison-Wesley, 1980.

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

[23] Charles E. Leiserson and James B. Saxe. Retiming Synchronous Circuitry. *Algorithmica*, 6:5–35, 1991.

[24] Alber A. Liddicoat and Michael J. Flynn. High-Performance Floating Point Divide. In *Euromicro Symposium on Digital System Design*, pages 354–361, Warsaw, Poland, September 2001.

[25] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, and Mark Horowitz. Smart Memories: A Modular Reconfigurable Architecture. In *28th Annual International Symposium on Computer Architecture*, pages 161–171, June 2000.

[26] Merriam-Webster, Inc. *Merriam-Webster's Collegiate Dictionary.* Springfield, MA, 10th edition, 2001.

[27] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, 1994.

[28] Ramdass Nagarajan, Karthikeyan Sankaralingam, Doug C. Burger, and Steve W. Keckler. A Design Space Evaluation of Grid Processor Architectures. In *34th Annual International Symposium on Microarchitecture*, pages 40–51, December 2001.

[29] Stuart F. Olberman and Michael J. Flynn. Implementing Division and Other Floating-Point Operations: A System Perspective. In *Proceedings of SCAN-95, International Symposium on Scientific Computing, Computer Arithmetic, and Validated Numerics*, Wuppertal, Germany, September 1995.

[30] James E. Smith. Decoupled Access/Execute Computer Architectures. *ACM Transactions on Computer Systems*, 2(4):289–308, November 1984.

[31] Robet Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[32] Michael B. Taylor. btl extension – for jedi masters. `http://cag.lcs.mit.edu/raw/memo/19/btl-advanced.html`.

[33] Michael B. Taylor. The Raw Processor Specification. `ftp://ftp.cag.lcs.mit.edu/pub/raw/documents/RawSpec99.pdf`.

[34] Michael B. Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. *IEEE Micro*, 22(2):25–36, March/April 2002.

[35] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffmann, Paul Johnson, Walter Lee, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Saman Amarasinghe, and Anant Agarwal. A 16-issue Multiple-program-counter Microprocessor with Point-to-point Scalar Operand Network. In *Proceedings of the IEEE International Solid-State Circuits Conference*, February 2003.

[36] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proceedings of the 2002 International Conference on Compiler Construction*, Grenoble, France, 2002.

[37] Sivan Toledo. A Survey of Out-of-Core Algorithms in Numerical Linear Algebra. In James Abello and Jeffrey Scott Vitter, editors, *External Memory Algorithms and Visualization*, pages 161–180. American Mathematical Society Press, Providence, RI, 1999.

[38] Michael Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1995.