LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Stream Algorithms and Architecture

Technical Memo
MIT-LCS-TM-636
March 26, 2003

Henry Hoffmann, Volker Strumpen, and Anant Agarwal

{hank,strumpen,agarwal}@cag.lcs.mit.edu

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Stream Algorithms and Architecture

Henry Hoffmann, Volker Strumpen, and Anant Agarwal
Massachusetts Institute of Technology

March 26, 2003

Abstract

Wire-exposed, programmable microarchitectures including Trips [11], Smart Memories [8], and Raw [13] offer an opportunity to schedule instruction execution and data movement explicitly. This paper proposes stream algorithms, which, along with a decoupled systolic architecture, provide an excellent match for the physical and technological constraints of single-chip tiled architectures. Stream algorithms enable programmed systolic computations for different problem sizes, without incurring the cost of memory accesses. To that end, we decouple memory accesses from computation and move the memory accesses off the critical path. By structuring computations in systolic phases, and deferring memory accesses to dedicated memory processors, stream algorithms can solve many regular problems with varying sizes on a constant-sized tiled array. Contrary to common sense, the compute efficiency of stream algorithms increases as we increase the number of processing elements. In particular, we show that the compute efficiency of stream algorithms can approach 100 % asymptotically, that is for large numbers of processors and appropriate problem size.

1 Introduction

Our goal is to show that microtechnology provides us with an opportunity to design single-chip parallel machines on which we may increase efficiency by increasing the number of processors for many important applications. Microtechnology is about to revolutionize the design of computer systems for the second time since the first single-chip microprocessor, Intel's 4004, was released in 1971. While the amount of transistors that fit onto a single chip has been growing steadily, only now, thirty years later, are we reaching the critical mass for realizing a general-purpose parallel microarchitecture on a single chip. Research prototypes such as Trips [11], Smart Memories [8], and Raw [13] 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 [4].

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 [10], parallel computation [6], and parallelizing compiler design [15] 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 [5], 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 [1] became programmable, so they could reap the benefits of systolic arrays without sacrificing flexibility. We believe that the significant area and energy efficiency of systolic arrays merit their reexamination in face of the architectural similarities to recent tiled microarchitectures.

A commonly accepted and generally applicable technique to overcome the specialization of a systolic array to a particular problem size is the ***simulation*** of multiple processing elements on one larger, more powerful systolic processor. Such a processor uses local memory to store a potentially unbounded amount of data. Thus, the local memories required to run large problems must also be large. Large memories use a load/store interface to cope with intrinsically large access times. Load and store operations do not contribute useful computation, however, and constitute a burden on the critical path. In contrast, small memories are fast and can be organized as register sets. Register accesses are integrated as operand accesses into machine instructions where they do not effect the critical path adversely. Thus, it seems worthwhile to abandon the simulation technique and apply the systolic method to large problem sizes while using a small amount of local memory only.

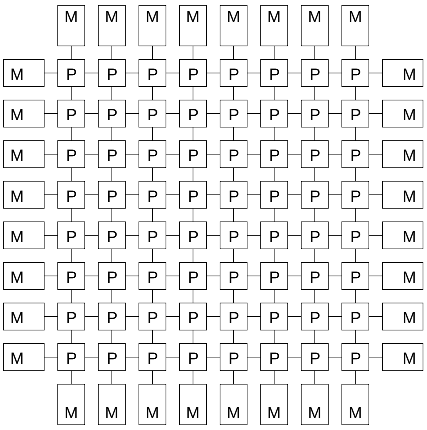

Figure 1: A decoupled systolic architecture (DSA) is an $R \times R$ array of compute processors (P) surrounded by $4R$ memory processors (M) as shown for $R = 8$. Compute processors use fast memory in form of a register set only. Each memory processor consists of a compute processor plus a memory interface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

In this article, we propose a new strategy for structuring parallel programs for tiled architectures, together with a resulting class of decoupled systolic algorithms that we call ***stream algorithms***. Stream algorithms allow tiled architectures to operate in a systolic fashion for many regular problems with variable problem sizes. Stream algorithms require augmenting a tiled compute array with a set of memory processors on the periphery of the array, and potentially off-chip, as illustrated in Figure 1. A stream algorithm solves a large problem by breaking it into smaller, systolic subproblems, and by storing input data and intermediate results in the peripheral memories. The input data are supplied to the compute processors from the memories in a continuous stream[1] via the network, thereby eliminating load/store memory accesses on the compute processors. Thus, stream algorithms decouple memory accesses from computation, and move the memory accesses off the critical path.

Stream algorithms combine the benefits of *decoupling*, the software version of the decoupled access/execute architecture [12], and *systolic algorithms*, the software version of systolic arrays [5]. Together, both features constitute an excellent match for the physical and technological constraints of future tiled microarchitectures. There is one catch, though. Since stream algorithms are structured in systolic phases, they do not achieve high compute efficiency unless the subproblems can be pipelined. Decoupling introduces yet another source of inefficiency, namely the memory processors that execute memory accesses rather than contributing useful computation. We show how stream algorithms amortize the loss of efficiency by using a large number of processors with an asymptotically insignificant amount of memory processors. We also identify a kernel of architectural features needed to implement a ***decoupled systolic microarchitecture*** for executing stream algorithms area efficiently.

The remainder of this paper is organized as follows. In Section 2 we introduce our decoupled systolic architecture. Section 3 defines our notion of stream algorithms. Then, we discuss five applications, a matrix multiplication in Section 4, a triangular solver in Section 5, an LU factorization in Section 6, a QR factorization in Section 7, and a convolution in Section 8. We show how to formulate these applications as stream algorithms, and argue why the resulting algorithms achieve optimal compute efficiency of 100 % asymptotically when executed on our decoupled systolic architecture.

2 A Decoupled Systolic Architecture

A decoupled systolic architecture (***DSA***) is a type of single-chip tiled architecture. We assume a fast network consisting of short wires connecting processors in a mesh topology as shown in Figure 1. Our DSA consists of an $R \times R$ array of ***compute processors***, and $4R$ ***memory processors*** on the periphery of the compute array. The peripheral memory processors are the distinguishing feature of DSA's. Each of the memory processors consists of a compute processor with additional local memory. We assume that the memory can deliver a throughput of one load or 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. 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 2, is a simple general-purpose programmable

[1] Our notion of a data stream is consistent with the colloquial sense. According to Webster [9], 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)."

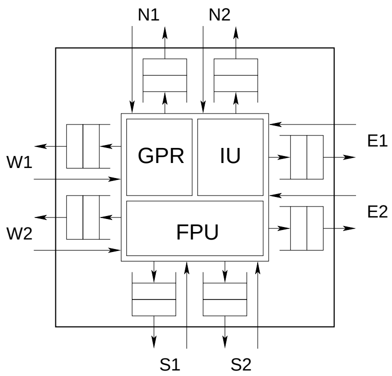

Figure 2: A compute processor contains a general-purpose register set (GPR), an integer unit (IU), and a floating-point unit (FPU) based on a multiply-and-add module. The processor connects via FIFO's to its four neighbors.

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. We do not include a larger local memory because of the intrinsic physical constraint that large memories have larger latencies than small ones. Instead, we use a register set with relatively small but fast memory. To focus our attention on the datapath, Figure 2 omits all of the control logic and a small instruction memory. We assume a single-issue, in-order pipelined FPU that allows us to issue one multiply-and-add operation per clock cycle.

Arguably the most important feature of a DSA is the design of its on-chip network. Our interconnect uses two prominent features of Raw's static network [13]. 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. We discuss these features in more detail in the following paragraphs.

As illustrated in Figure 2, we use blocking FIFO's to connect and synchronize 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, we prefer 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.[2] 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.

Our decoupled systolic architecture uses a wide instruction word to schedule multiple, simultaneous data movements across the network, between the functional units and the

[2] The tight integration with the processor pipeline is a key design aspect of the Raw architecture [13], which earlier register-mapped network architectures including the Connection Machine CM2 [3] and Warp [1] lack.

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
```

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 our 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 our DSA, we assume that three data movements can be specified within the route part of a single instruction.

DSA's may be implemented as a virtual machine on top of a tiled architecture. We have experimented with this idea on Raw, which can be viewed as an architectural superset of our DSA. Since every Raw processor has local memory and can be viewed as a memory processor, we simulate our compute processors simply by ignoring the available local memory. In that respect, Raw is a functional, but not area-efficient DSA. Raw has separate switch and compute processors on each tile, each with their own instruction stream. We splice the instruction stream of each of our DSA processors into a Raw processor stream, a Raw switch stream, plus synchronization primitives. Another difference is that Raw's floating-point unit is not based on a multiply-and-add module. Thus, the `fma` instruction of our DSA requires two instructions on Raw.

3 Stream Algorithms

In this section we introduce decoupled systolic algorithms, nicknamed stream algorithms, and a set of conditions for which we can increase efficiency by increasing the number of processors such that the compute efficiency approaches 100 %. Alternatively, we may view stream algorithms as the product of a program-structuring methodology. We identify five design principles for stream algorithms:

1. We ***reduce the instruction count on the critical path*** of a computation by abandoning load and store instructions on the compute processors.

2. We use ***systolic designs***, because they are well suited for parallel machines with local interconnect structure and match our architecture with fast but short wires.

3. We ***decouple memory accesses from computation*** by dedicating processors to one of the two tasks. This idea is motivated by the Decoupled Access/Execute Architecture [12].

4. We 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 ***M=o(P)***.

5. We ***partition problems*** into decoupled systolic algorithms, and use the M memory processors for temporary storage.

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, we can actually increase the efficiency for larger numbers of processors and large problems. We emphasize this observation 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,[3] *let $P(R)$ be the number of compute processors and $M(R)$ the number of memory processors. We say the algorithm is* ***decoupling efficient*** *if and only if*

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

Informally, decoupling efficiency expresses that the number of memory processors becomes insignificant relative to the number of compute processors as we increase 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 we implement an algorithm on $P = R^2$ compute processors. If we 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 $P = \Theta(R^2)$, we 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 systolic algorithms per se are independent of a particular architecture. Note, however, that the DSA shown in Figure 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. We determine 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 $P(R) + M(R)$:

$$E(N, R) = \frac{C(N)}{T(N, R) \cdot (P(R) + M(R))}. \tag{1}$$

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, we may relate the problem size N and

[3] We use the network size R 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.

network size R via a real-valued σ such that $N = \sigma R$. Substituting σR for N in Equation 1, we define compute efficiency by means for the following condition.

Definition 2 ***(Compute-Efficiency Condition)***
We call an algorithm with problem size N ***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 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 we operate a compute array without any memory processors it is a systolic array. We are interested in the case where $M > 0$, however, and 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 we require that $N \gg R$, which we view as a realistic assumption since using a very large network implies that we intend to solve a very large problem. For $\sigma = 1$, the problem size matches the network size, and we operate the network as a systolic array. Since decoupling-efficient stream algorithms use an asymptotically smaller number of memory processors than compute processors, we may view stream algorithms as a subset of systolic algorithms with a restricted number of inputs and outputs. Inversely, we may view a systolic algorithm as a special case of a stream algorithm that is distinguished by $\sigma = 1$ in $N = \sigma R$. We discuss the trade-off between N and R during our discussion of stream algorithms below.

Before presenting concrete examples of stream algorithms, we outline our general ***stream-structuring methodology***, 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, we start 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, we find that recursive formulations and partitioning methods like those developed for out-of-core algorithms [14] can be helpful. To simplify the design of the systolic algorithm, *retiming* [7] may be used. It allows us to start the design with a semi-systolic algorithm, which we can transform automatically into a systolic algorithm if one exists [6]. 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.

Decoupling: Our goal is to move the memory accesses off the critical path. To this end, we have to 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 we integrate a fully-fledged compute processor into the memory processor of our stream architecture.

Efficiency Analysis: After partitioning and decoupling, we have designed a stream algorithm. To qualify as a compute-efficient stream algorithm, however, we also require 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 we must show that $E(\sigma, R)$ approaches 1 for large values of R. Meeting the compute-efficiency condition requires that we order the subproblems for optimal pipelining on the compute array. Experience shows that we may need to iterate over the partitioning and decoupling steps until a compute-efficient solution is found.

Let us emphasize the concept of a stream algorithm by highlighting 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 that 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 Matrix Multiplication

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

$$c_{ij} \quad = \quad \sum_{k=1}^{N} a_{ik} \cdot b_{kj}, \qquad (2)$$

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

Partitioning

We use a block-recursive partitioning for the matrix multiplication. We recurse along the rows of A and the columns of B:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \end{pmatrix}. \tag{3}$$

For each of the matrices C_{ij} we have $C_{ij} = A_{i1}B_{1j}$, 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.

Decoupling

We begin by observing that each product element c_{ij} can be computed independently of all others by means of Equation 2. In addition, Equation 3 allows us to stream entire rows of A and entire columns of B through the compute processors. Furthermore, we partition a problem of size $N \times N$ until the C_{ij} are of size $R \times R$ and fit into our array of compute processors. We implement the resulting subproblems as systolic matrix multiplications, illustrated in Figure 3 for $N = R = 2$. Rows of A flow from the left to the right, and columns of B from the top to the bottom of the array.

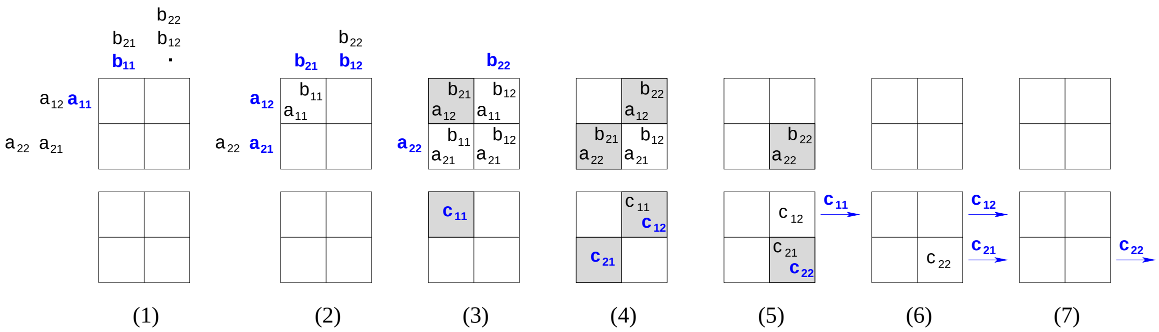

Figure 3: Seven time steps of a systolic matrix multiplication $C = A \cdot B$ for 2×2 matrices. Each box represents a compute processor. Values entering, leaving, or being generated in the array are shown in bold face. Shaded boxes mark the completion of an inner product. We split the data flow of the operands and products into the top and bottom rows.

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, we use 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, we use $P = R^2$ compute processors and $M = 2R$ memory processors. Figure 4 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 of rows of A and columns of B into the compute processors. First, we compute C_{11} by streaming $\{A(1,:),\ A(2,:)\}$ and $\{B(:,1),\ B(:,2)\}$ through the array. Second, we stream $\{A(1,:),\ A(2,:)\}$ against $\{B(:,3),\ B(:,4)\}$, third, $\{A(3,:),\ A(4,:)\}$ against $\{B(:,1),$

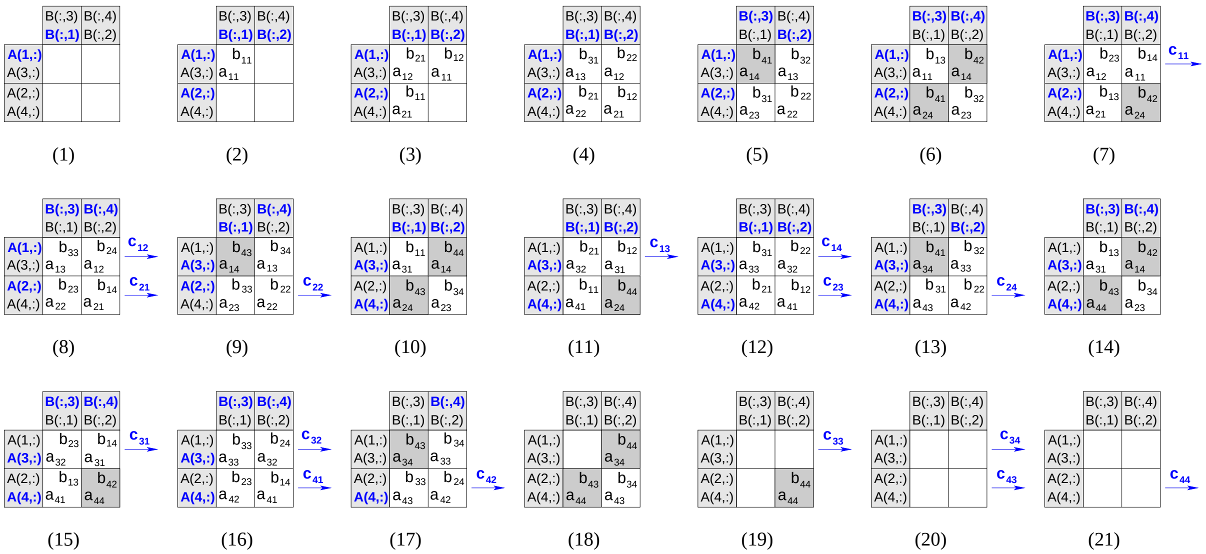

Figure 4: Data flow of a compute-efficient matrix multiplication $C = A \cdot B$ for 4×4 matrices on 2×2 compute processors. Shaded boxes on the periphery mark memory processors, and indicate the completion of an inner-product otherwise.

$B(:,2)\}$, and finally $\{A(3,:), A(4,:)\}$ against $\{B(:,3), B(:,4)\}$. As a result, we compute 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, we may have to invest another R memory processors for a total of $M = 3R$. In any case, we have $P = \Theta(R^2)$ and $M = \Theta(R)$, and hence $M = o(P)$. We conclude that the structure of our matrix multiplication is decoupling efficient.

Note that we 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.

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, we pipeline 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 4), the total number of time steps required by this computation is

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

According to Equation 1, the floating-point efficiency of our 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, we obtain

$$E_{mm}(\sigma, R) = \frac{\sigma^3}{\sigma^3 + 3} \cdot \frac{R}{R + 2} \tag{4}$$

for the efficiency. Consider each of the two product terms independently. Term $\sigma^3/(\sigma^3 + 3)$ approaches 1 for large values of σ, that is if the problem size N is much larger than the network size R. On the other hand, term $R/(R + 2)$ approaches 1 for large network sizes R. If we assume a constant value $\sigma \gg 1$, we find that the efficiency of the matrix multiplication increases as we increase the network size, and approaches the optimal floating-point efficiency of 100 % asymptotically. We 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, we want to maximize both terms in Equation 4. Thus, given a problem size N, to increase the first term, we want to increase $\sigma = N/R$ and, hence, decrease R. On the other hand, to maximize the second term, we want to increase R. To determine a good value R for implementing a DSA, let us consider some absolute numbers. For example, if $N = R$, that is $\sigma = 1$, we have 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 $\sigma = 8$, we have

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

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.

5 Triangular Solver

A triangular solver computes the solution x of a linear system of equations $Ax = b$ assuming that matrix A is triangular. Here is 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***:

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

We are interested in triangular solvers as building blocks of other algorithms including an LU factorization. In particular, we are interested in 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.

Partitioning

We partition the lower-triangular system of linear equations with multiple right-hand sides recursively according to Equation 5. Matrices A_{11} and A_{22} are lower triangular.

$$\begin{pmatrix} A_{11} & 0 \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{pmatrix} = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \tag{5}$$

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

$$\begin{aligned} A_{11}X_{11} &= B_{11} && (6) \\ A_{11}X_{12} &= B_{12} && (7) \\ B'_{21} &= B_{21} - A_{21}X_{11} && (8) \\ B'_{22} &= B_{22} - A_{21}X_{12} && (9) \\ A_{22}X_{21} &= B'_{21} && (10) \\ A_{22}X_{22} &= B'_{22} && (11) \end{aligned}$$

First, we compute the solution of the lower-triangular systems in Equations 6 and 7, yielding X_{11} and X_{12}. We use these solutions subsequently to update matrices B_{21} and B_{22} in Equations 8 and 9, producing B'_{21} and B'_{22}. We could compute the matrix subtraction in Equations 8 and 9 on the compute processors of the array. However, we can save the associated data movement 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 10 and 11. Solving these systems yields X_{21} and X_{22}. Thus, Equations 6–11 define a recursive algorithm for solving the lower-triangular system of Equation 5. The recursion reduces the problem of solving a lower-triangular system of linear equations into four smaller lower-triangular systems of linear equations, plus two matrix multiplications that we have discussed in Section 4 already.

Decoupling

To arrive at a decoupled design, we observe 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 5 depicts the systolic algorithm for our lower-triangular solver. We stream rows of A from the left to the right and columns of B from the top to the bottom of the compute array, while 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 we may permute the columns of B arbitrarily, provided we preserve the staggered data movement. We can also use the systolic design of Figure 5 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.

We illustrate the systolic algorithm by describing the computation of element $x_{31} = (b_{31} - a_{31}x_{11} - a_{32}x_{21})/a_{33}$. We begin with time step 4 in Figure 5. 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 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, we need $3R$ memory processors on the periphery of the compute array to buffer matrices A, B, and X. Figure 6 shows the computation of X_{11} and X_{21} by means of Equations 6, 8, and 10. As implied by this figure, we use R memory processors to store the rows of A, and R memory processors for the columns of B and X, respectively. Thus, for a decoupled systolic lower-triangular solver, we require $P = R^2$ compute processors and $M = 3R$ memory processors, meeting our decoupling-efficiency condition $M = o(P)$.

Unlike the matrix multiplication, the factorization of the triangular solver does not produce identical subproblems. Therefore, we are faced with the additional challenge of finding an efficient composition of these subproblems. Although we can pipeline the subproblems, we cannot avoid idle cycles due to data dependencies and the heterogeneity of the computations in Equations 6–11. However, we can minimize the loss of cycles by grouping independent computations of the same type, and pipelining those before switching to another group. For example, we can group and pipeline the computations of Equations 6 and 7, then Equations 8 and 9, and finally Equations 10 and 11. If we unfold the recursion all the way to the base case of $R \times R$ subproblems, we find that the best schedule is equivalent to a block-iterative ordering of the subproblems.

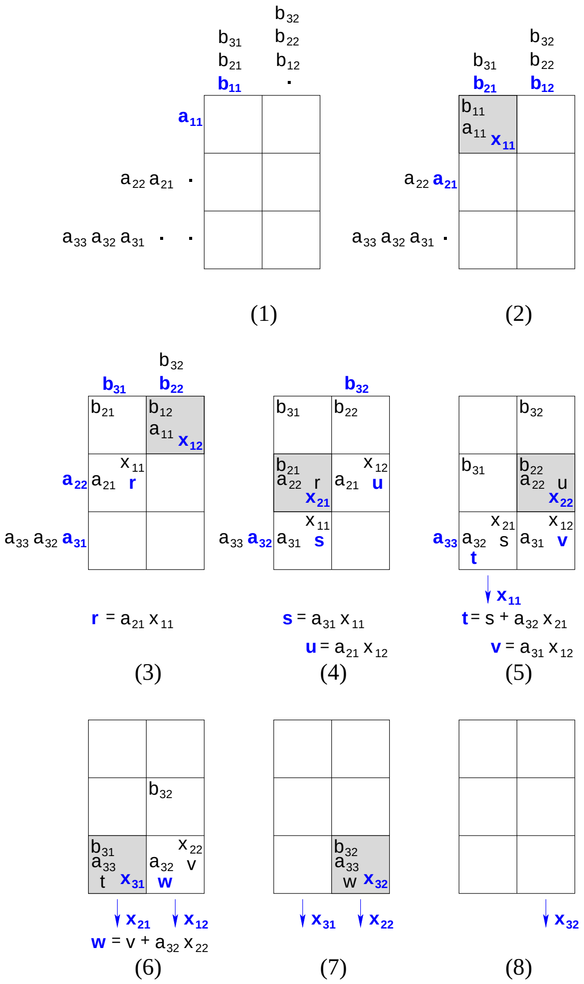

Figure 5: Systolic lower-triangular solver for $N = 3$ and two right-hand sides.

Efficiency Analysis

We compute the efficiency of our lower-triangular solver according to Equation 1. The number of floating-point multiply-and-add operations is $C(N) = N^3/2$, counting each division as a multiply-and-add operation.

As mentioned above, the crux for an efficient schedule is to order the computations in Equations 6–11 such that two subsequent systolic algorithms can be overlapped. For the matrix multiplication, finding a perfect overlap is relatively easy, because the there is only one systolic algorithm. For the lower-triangular solver, we illustrate the search for a good schedule and the corresponding efficiency analysis by means of the example in Equation 12, where matrices A, X, and B consist of $\sigma \times \sigma$ blocks, each block is of dimension $R \times R$, and $\sigma = 4$.

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

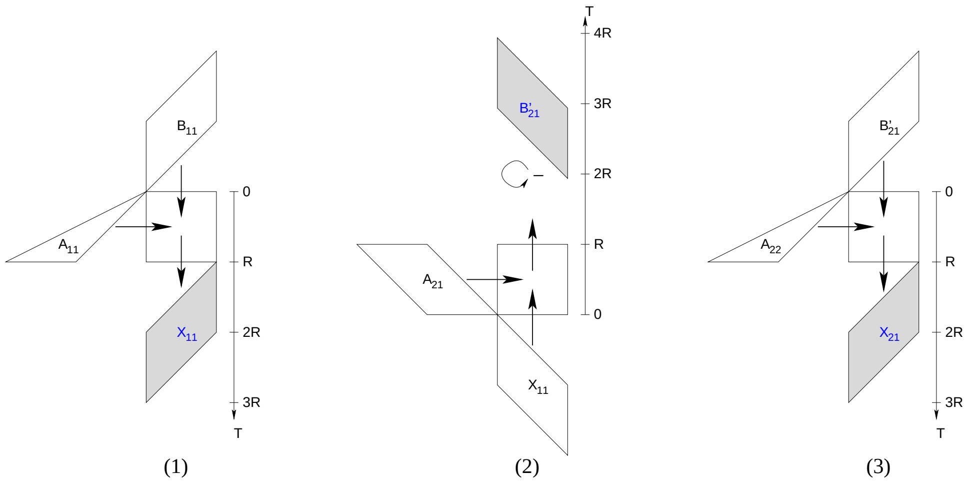

Figure 6: Phases of a decoupled systolic lower-triangular solver on an $R \times R$ array of compute processors. In phase 1 we solve Equation 6 for X_{11}. In phase 2 we update B_{21} according to Equation 8. While the matrix multiplication is executed on the compute processors, the matrix subtraction is performed on the memory processors as they receive each individual result of $A_{21}X_{11}$. Finally, in phase 3 we solve Equation 10 for X_{21}. The shapes of the matrix areas indicate how the rows and columns enter the compute array in a staggered fashion.

Recall that the computations of the individual columns and, thus, column blocks of X are independent. Therefore, we may sequence the systolic computations across rows to maximize overlap. In the 2×2 partitioning of Figure 6, the computation of a column block of X consists of two solver computations and one update operation. The interleaving of two such computations yields the sequence of Equations 6–11. Now, consider column block i of the 4×4 example in Equation 12 with the following sequence of operations. First, solve $A_{11}X_{1i} = B_{1i}$ for X_{1i}. Second, update $B'_{2i} = B_{2i} - A_{21}X_{1i}$. Third, solve $A_{22}X_{2i} = B'_{2i}$ for X_{2i}. Fourth, update $B'_{3i} = B_{3i} - A_{31}X_{1i} - A_{23}X_{2i}$, which involves two update operations. Fifth, solve $A_{33}X_{3i} = B'_{3i}$ for X_{3i}. Sixth, update $B'_{4i} = B_{4i} - A_{41}X_{1i} - A_{42}X_{2i} - A_{43}X_{3i}$, involving three update operations. Finally, solve $A_{44}X_{4i} = B'_{4i}$ for X_{4i}. We observe that for increasing σ, the number of solvers increases quadratically while the number of update operations increases cubically.

Since the update operations are little more than matrix multiplications, we may pipeline and overlap as many of them as possible, resembling our stream-structured matrix multiplication. Thus, we use a block iterative schedule that iterates over row blocks. For each row block i, we schedule the solver computations of an entire row i with maximal overlap. There are σ solvers for each row block, which require σR time steps plus $2R$ time steps for starting and draining the pipeline. Then, we compute and overlap all update operations associated with X_{ij}. For each X_{ij}, there are $\sigma - i$ update operations, resulting in $\sigma(\sigma - i)$ update operations associated with row i. These operations require $\sigma(\sigma - i)R$ time steps plus $3R$ time steps to start and drain the pipeline. We may save another $2R$ time steps by overlapping the first update operation with the last solver computation and the first solver computation of the next row block with the last update operation of the previous row block. The number of time steps for an $N \times N$ lower-triangular solver with $\sigma \times \sigma$ blocks of size

$R \times R$ is then:

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

We find that, 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 1, the floating-point efficiency of our compute-efficient lower-triangular solver is then

$$E_{lts}(\sigma, R) = \frac{\sigma^3}{\sigma^3 + \sigma^2 + 6\sigma - 2} \cdot \frac{R}{R+3}. \tag{13}$$

Analogous to Equation 4 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. For $\sigma = 1$, the problem reduces to a single systolic lower-triangular solver, and we obtain an efficiency of

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

The efficiency increases when we increase 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, we have $E_{lts}(R) \approx R/(R+3)$, and we achieve more than 90 % efficiency for $R > 27$.

6 LU Factorization

We wish to factor 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, for all diagonal elements of $L = (l_{ij})$ we require that $l_{ii} = 1$, where $1 \leq i \leq N$.

Partitioning

We partition the LU factorization according to Equation 14. Matrices L_{11} and L_{22} are lower triangular, while matrices U_{11} and U_{22} are upper triangular.

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{pmatrix} \tag{14}$$

This partitioning results in a series of smaller problems:

$$A_{11} = L_{11}U_{11} \tag{15}$$

$$A_{12} = L_{11}U_{12} \tag{16}$$

$$A_{21} = L_{21}U_{11} \tag{17}$$

$$A'_{22} = A_{22} - L_{21}U_{12} \tag{18}$$

$$A'_{22} = L_{22}U_{22} \tag{19}$$

First, we compute L_{11} and U_{11} by factoring A_{11} according to Equation 15. We use the results to solve Equations 16 and 17 for U_{12} and L_{21} respectively. Then, we update A_{22} according to Equation 18, which produces A'_{22}. As with the triangular solver, the matrix subtraction required by this step can be performed on the memory tiles. Finally, we use A'_{22} to compute L_{22} and U_{22}. The recursive formulation due to Equations 15–19 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.

Decoupling

We begin the derivation of a decoupled design 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 shows the progress of our systolic LU factorization. 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 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}. As a concrete example, we discuss 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}. We begin at time step 6 of Figure 7. Processor p_{21} receives u_{11} from above and uses it to compute element l_{21}, which 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, we reduce a problem of size $N \times N$ recursively until the subproblems fit into an $R \times R$ array of compute processors. Figure 8 illustrates the data movement of the matrices when computing five systolic subproblems according to Equations 15–19. We need 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, our decoupled systolic LU factorization requires $P = R^2$ compute processors and $M = 3R$ memory processors. We observe that $M = o(P)$, and the structure of our LU factorization is decoupling efficient.

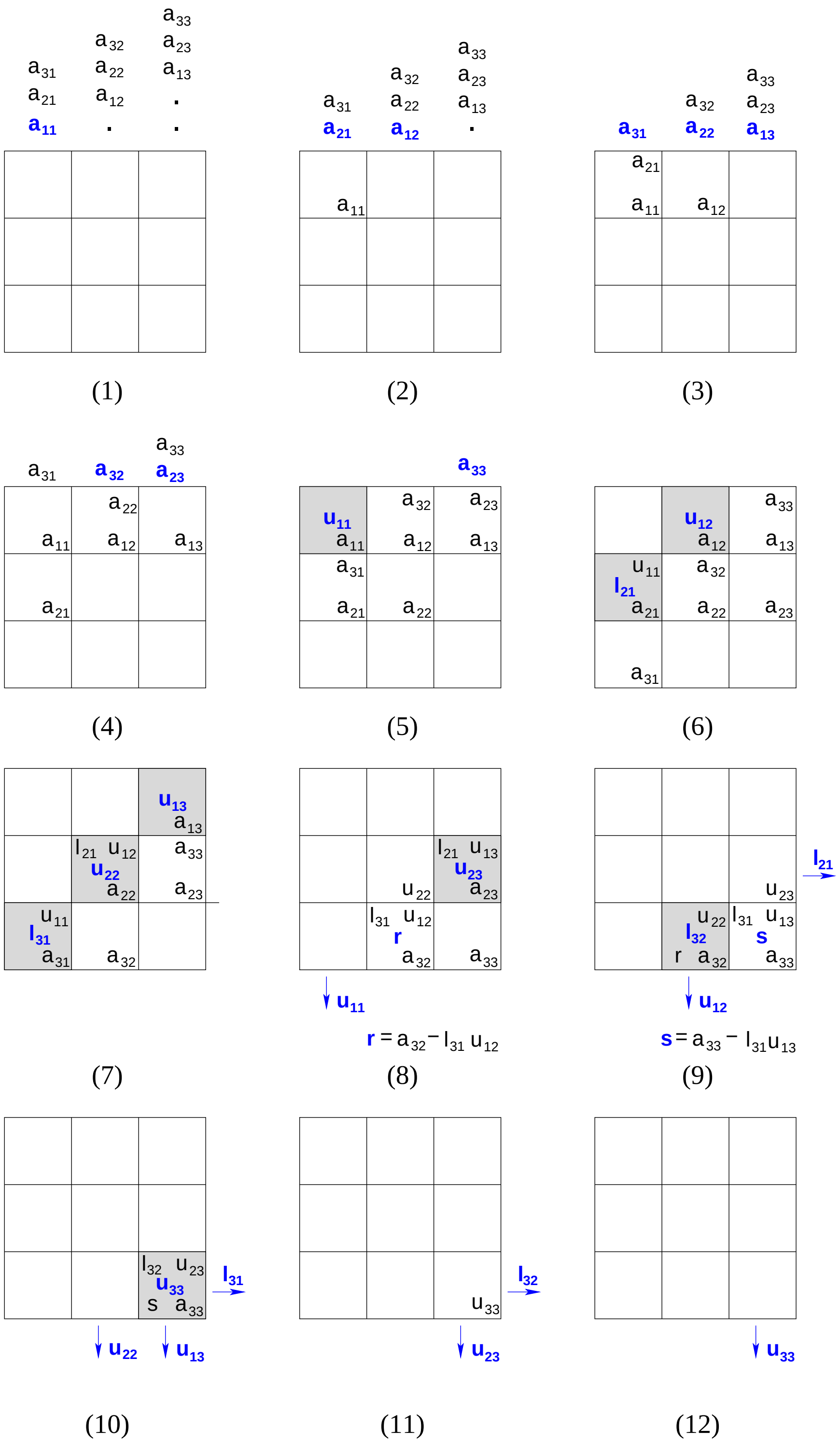

Figure 7: Systolic LU factorization for $N = 3$.

Similar to our lower-triangular solver, the partitioning due to Equations 15–19 produces a set of heterogeneous subproblems. To obtain the most efficient composition of the subproblems, we 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 three-fold loop of the LU factorization, we can solve Equation 15 for each pivot block, pipeline the solvers of Equations 16 and 17 for the pivot row and column, and update the lower right matrix by pipelining the matrix multiplications of Equation 18. The computation of Equation 19 corresponds to the factorization of the next pivot block.

Efficiency Analysis

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

To find an efficient schedule for the LU factorization, we apply the methodology that we introduced for the lower-triangular solver. We unfold the recursive partitioning to the leaves, where each subproblem has size $R \times R$, and group the individual systolic computations so as to maximize their overlap. Our block-iterative schedule resembles the standard 3-fold loop of the LU factorization. For each pivot block k, we factor A_{kk} into L_{kk} and U_{kk}, apply lower and upper triangular solvers to compute the L_{ik} and U_{kj}, and update the lower right matrix. The systolic LU factorization requires $4R$ time steps. The systolic solvers for computing the $(\sigma - k)$ matrices U_{kj} use $(\sigma - k)R$ time steps plus $2R$ time steps to start and drain the pipeline. The computation of the $(\sigma - k)$ matrices L_{ik} also requires $(\sigma - k)R$ time steps, but $3R$ time steps to start and drain the pipeline, because matrices A and U enter the array from the top and the bottom, cf. Figure 8(3), rather than the top and the right, cf. Figure 8(2). Updating the $(\sigma - k) \times (\sigma - k)$ lower right matrix blocks uses $(\sigma - k)^2 R$ time steps plus $3R$ time steps for starting and draining the pipeline. We may save a few time steps by overlapping the first and last operations of these phases as follows. We can overlap the first lower-triangular solver with the preceding LU factorization by R time steps, the first upper-triangular with the last preceding lower-triangular solver by $2R$ time steps, the first update operation and the last preceding upper-triangular solver by R time steps, and the LU factorization succeeding the last update operation by $2R$ time steps. The number of time steps of an $N \times N$ LU factorization with $\sigma \times \sigma$ blocks of size $R \times R$ is then:

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

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

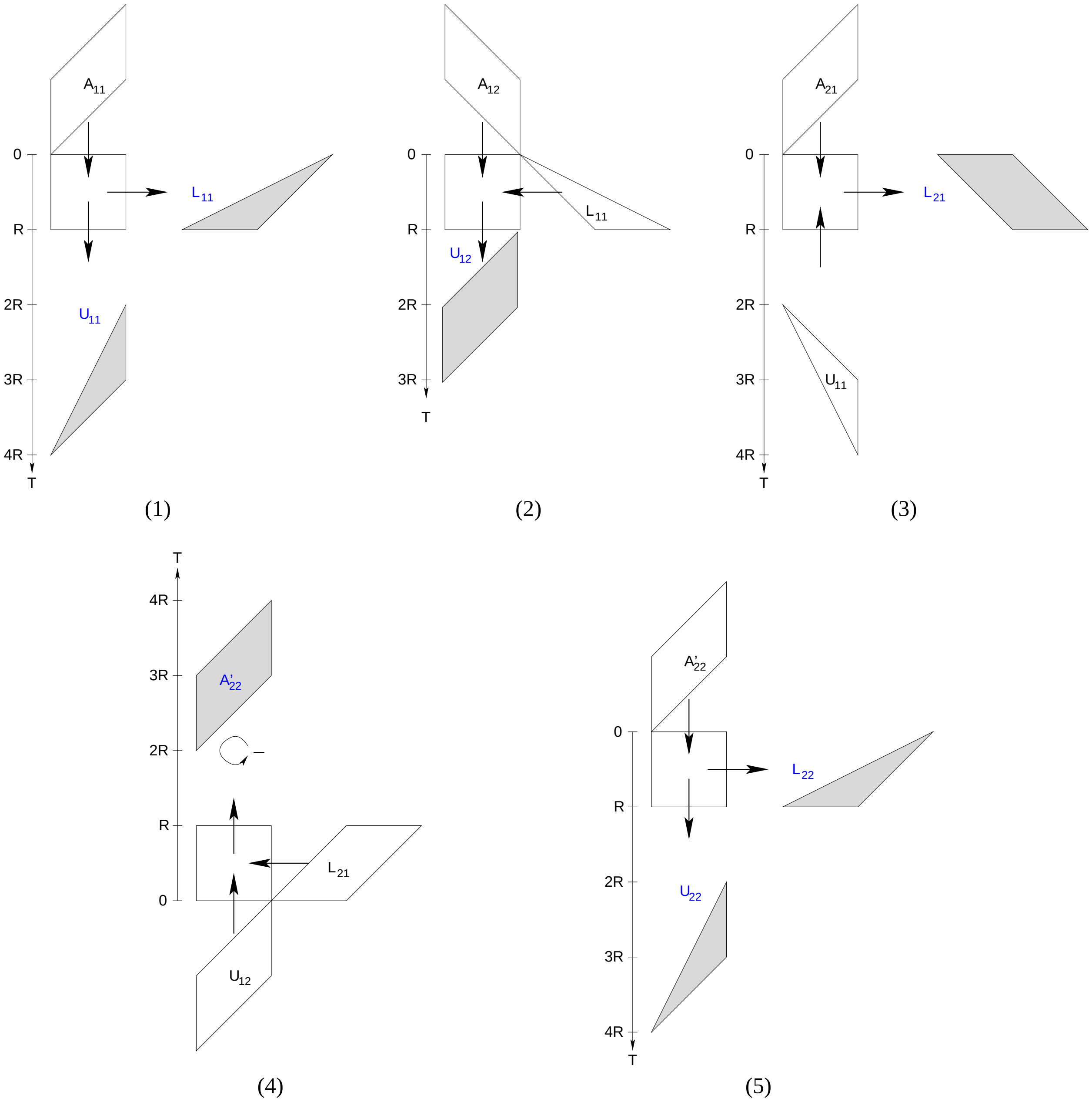

Figure 8: Phases of stream-structured LU factorization on an $R \times R$ array of compute processors, with $N = 2R$. In phase 1 we factor A_{11} into L_{11} and U_{11} according to Equation 15. In phase 2 we solve Equation 16 for U_{12}. In phase 3 we solve Equation 17 for L_{21}. In phase 4 we compute A'_{22} according to Equation 18, performing the matrix subtraction on the memory tiles. Finally, in phase 5 we factor A'_{22} according to Equation 19. The shapes of the matrices indicate how the rows and columns enter the compute array in a staggered fashion.

Using a network of size R, $P = R^2$ compute processors, and $M = 3R$ memory processors, the floating-point efficiency of our LU factorization according to Equation 1 is

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

Analogous to our treatment of E_{mm} and E_{lts} in Equations 4 and 13, we 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 we obtain 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 our LU factorization on our DSA approaches the optimal value of 100 %. As for the triangular-solver, when $\sigma \gg 1$ we have $E_{lu}(R) \approx R/(R+3)$, and we achieve more than 90 % efficiency for $R \geq 27$.

7 QR Factorization

We want to factor 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 $A = QR$. We assume that the factorization is based on fast Givens rotations [2]. The application of a fast Givens rotation, expressed as a matrix multiplication, annihilates an element of A. We call the $M \times M$ matrix $G(i,j)$ a ***fast Givens transformation***, if it places value zero in element (i,j) of the product $G(i,j)^T \cdot A$, and restrict $G(i,j) = (g_{\hat{\imath}\hat{\jmath}})$ to the form

$$G(i,j) \quad = \quad \begin{array}{cc} & \begin{array}{ccccccc} & & j & & i & & \end{array} \\ \begin{array}{c} \\ \\ j \\ \\ i \\ \\ \end{array} & \begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & \alpha & & \\ & & & \ddots & & & \\ & & \beta & & 1 & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{pmatrix} \end{array}, \qquad g_{\hat{\imath}\hat{\jmath}} = \begin{cases} 1, & \hat{\imath} = \hat{\jmath} \\ \alpha, & \hat{\imath} = j \wedge \hat{\jmath} = i \\ \beta, & \hat{\imath} = i \wedge \hat{\jmath} = j \\ 0, & \text{otherwise.} \end{cases} \tag{20}$$

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 us to compute α and β without square root operations. We initialize D such that $d_i = 1$ for $1 \leq i \leq M$.

We apply a series of fast Givens transformations to A to create an upper triangular matrix. Since transformation $G(i,j)$ annihilates element a_{ij}, we apply $MN - N^2/2 - N/2$ fast Givens transformations with $1 \leq j \leq N$ and $j < i \leq M$ to triangularize matrix A. Once a particular sequence of fast Givens transformations is chosen, the computation of D and Q must observe this order. We pick 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, we apply 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, proceed 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 $(AB)^T = B^T A^T$, we can write this product in compact form as[4]

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

Finally, let us discuss the role of diagonal matrix D briefly. By construction of the fast Givens transformation, we have $G^T G = D$. Since D is diagonal, we may split D such that $D = D^{1/2} D^{1/2}$. Then we obtain $D^{-1/2} G^T G D^{-1/2} = (GD^{-1/2})^T (GD^{-1/2}) = I$, and notice that $GD^{-1/2}$ is orthogonal. Thus, we may rewrite $G^T A = U$ 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.

Partitioning

We partition the QR factorization according to Equation 21 for $M \geq N$. Without loss of generality, we restrict our discussion to the case where $M = 3/2N$, such that matrices A_{ij} and R_{ij} in Equation 21 are $N/2 \times N/2$ matrices.

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

This partitioning allows us to formulate the QR factorization as a series of three subproblems, defined by Equations 22–24.

$$\begin{pmatrix} A_{11} \\ A_{21} \\ A_{31} \end{pmatrix} \;=\; Q_1 \begin{pmatrix} R_{11} \\ 0 \\ 0 \end{pmatrix} \tag{22}$$

[4] We introduce the convention 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 we interpret products as right-associative and the matrix multiplication as left-associative to determine the sequence uniquely. For example,

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

$$\begin{pmatrix} R_{12} \\ A'_{22} \\ A'_{32} \end{pmatrix} = Q_1^T \begin{pmatrix} A_{12} \\ A_{22} \\ A_{32} \end{pmatrix} \tag{23}$$

$$\begin{pmatrix} A'_{22} \\ A'_{32} \end{pmatrix} = \hat{Q}_2 \begin{pmatrix} R_{22} \\ 0 \end{pmatrix} \tag{24}$$

Equations 22–24 enable us to compute the QR factorization by solving three smaller problems. First, we triangularize columns $1, \ldots, N/2$ of A according to Equation 22. As a result, we obtain R_{11} and a sequence of fast Givens transformations associated with matrix Q_1. Next, we update columns $N/2+1, \ldots, N$ of A according to Equation 23, 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, we triangularize columns $N/2+1, \ldots, N$ according to Equation 24, 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 25–27:

$$Q_1 = \left(\prod_{j=1}^{N/2} \prod_{i=j+1}^{M} G(i,j) \right) \hat{D}^{-1/2} \tag{25}$$

$$Q_2 = \begin{pmatrix} I & 0 & 0 \\ 0 & \multicolumn{2}{c}{\hat{Q}_2} \\ 0 & & \end{pmatrix} = \left(\prod_{j=N/2+1}^{N} \prod_{i=j+1}^{M} G(i,j) \right) \tilde{D}^{-1/2} \tag{26}$$

$$Q = Q_1 Q_2 = \left(\prod_{j=1}^{N} \prod_{i=j+1}^{M} G(i,j) \right) \hat{D}^{-1/2} \tilde{D}^{-1/2}, \tag{27}$$

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.

According to common practice, we do not compute the intermediate forms Q_1 and Q_2 of matrix Q in order to triangularize matrix A. Instead, we utilize the sparse structure of the fast Givens transformation to operate with a highly efficient representation. Recall from the definition of $G(i,j)$ in Equation 20 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. We say that we ***premultiply*** a matrix A with fast Givens transformation $G(i,j)$ when forming the product $G(i,j)^T \cdot A$, and we ***postmultiply*** matrix A 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 9.

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. We note that both premultiplications

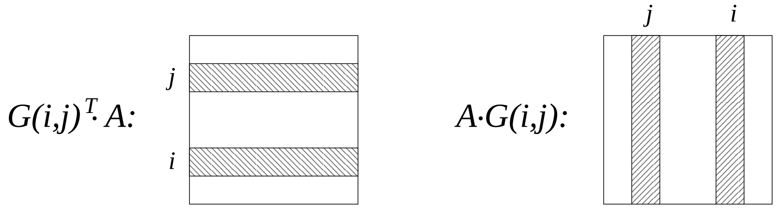

Figure 9: The product of premultiplication $G(i,j)^T \cdot A$ differs from A in rows i and j only, while the product of postmultiplication $A \cdot G(i,j)$ differs from A in columns i and j only.

and postmultiplications offer various degrees of parallelism. Specifically, we will 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 21 permits us to express the computation of matrix R of the QR factorization as two smaller QR factorizations of Equations 22 and 24 and a sequence of premultiplications with the fast Givens transformations associated with Q_1^T according to Equation 23. 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.

Decoupling

Our 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 ***systolic Givens computation*** by triangularizing an $M \times R$ matrix, where $M \geq R$; cf. Equation 22 and, analogously, Equation 24. 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 23 as a ***systolic premultiplication*** of an $M \times R$ matrix. The third algorithm computes matrix Q by means of ***systolic postmultiplications*** according to Equations 25–27 on $R \times M$ matrices.

Figures 10 and 11 illustrate the systolic Givens computation for triangularizing an $M \times R$ matrix A, where $M \geq R$. 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[5] 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

[5]We denote the sequence of n updates of diagonal element d_i 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}, $u_{ij}^{(1)}$, $u_{ij}^{(2)}$, …, $u_{ij}^{(n)}$, $r_{ij} = \delta_i u_{ij}^{(n)}$.

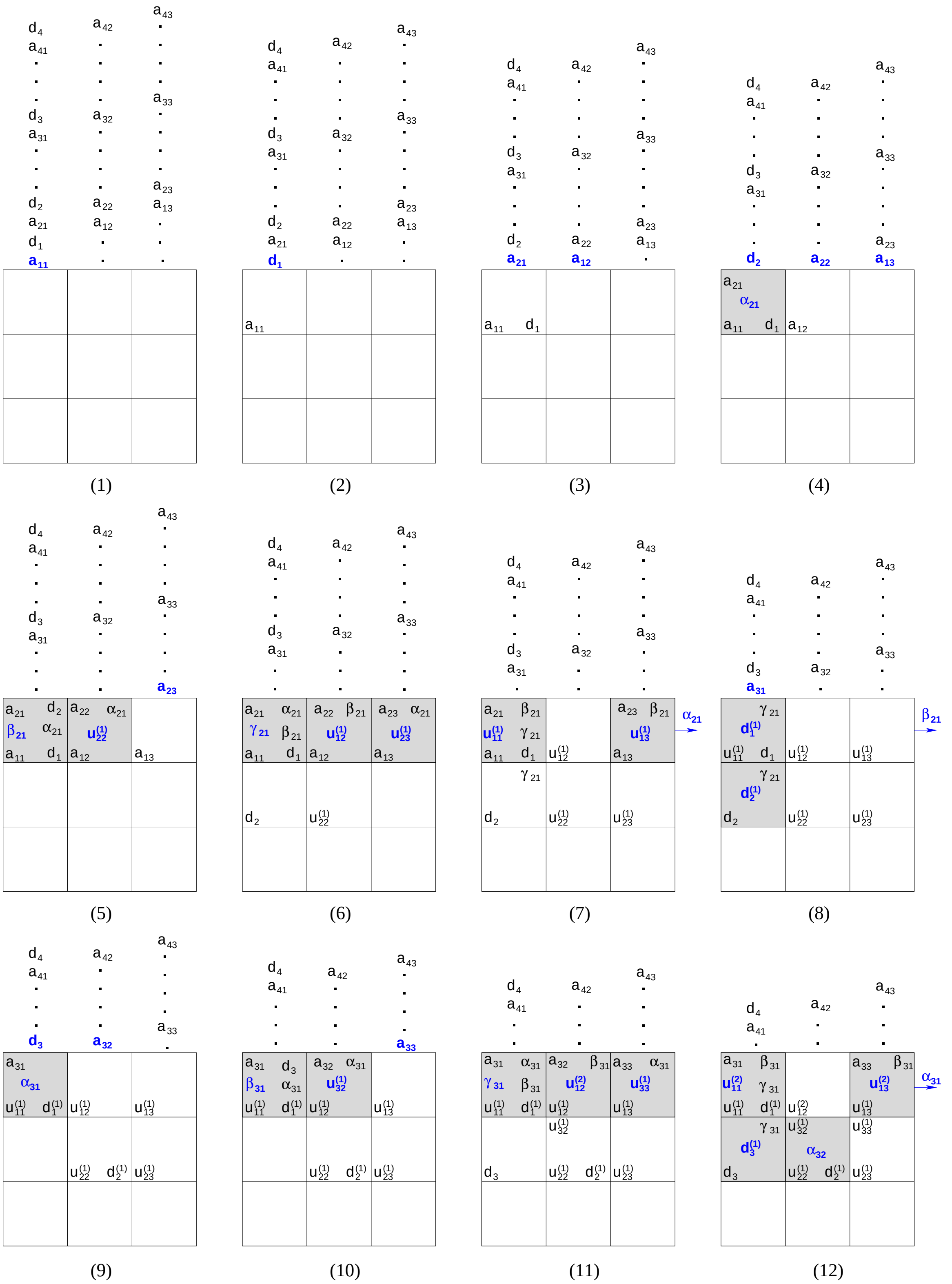

Figure 10: Systolic Givens computation for an $M \times R$ matrix, where $M \geq R$. We show the triangularization of a 4×3 matrix A, such that $R = D^{-1/2}\ G(4,3)^T\ G(4,2)^T\ G(3,2)^T\ G(4,1)^T\ G(3,1)^T\ G(2,1)^T \cdot A$. The fast Givens transformation $G(i,j)$ is represented by the pair $(\alpha_{ij}, \beta_{ij})$, and $\delta_i = d_i^{-1/2}$. [continued in Figure 11]

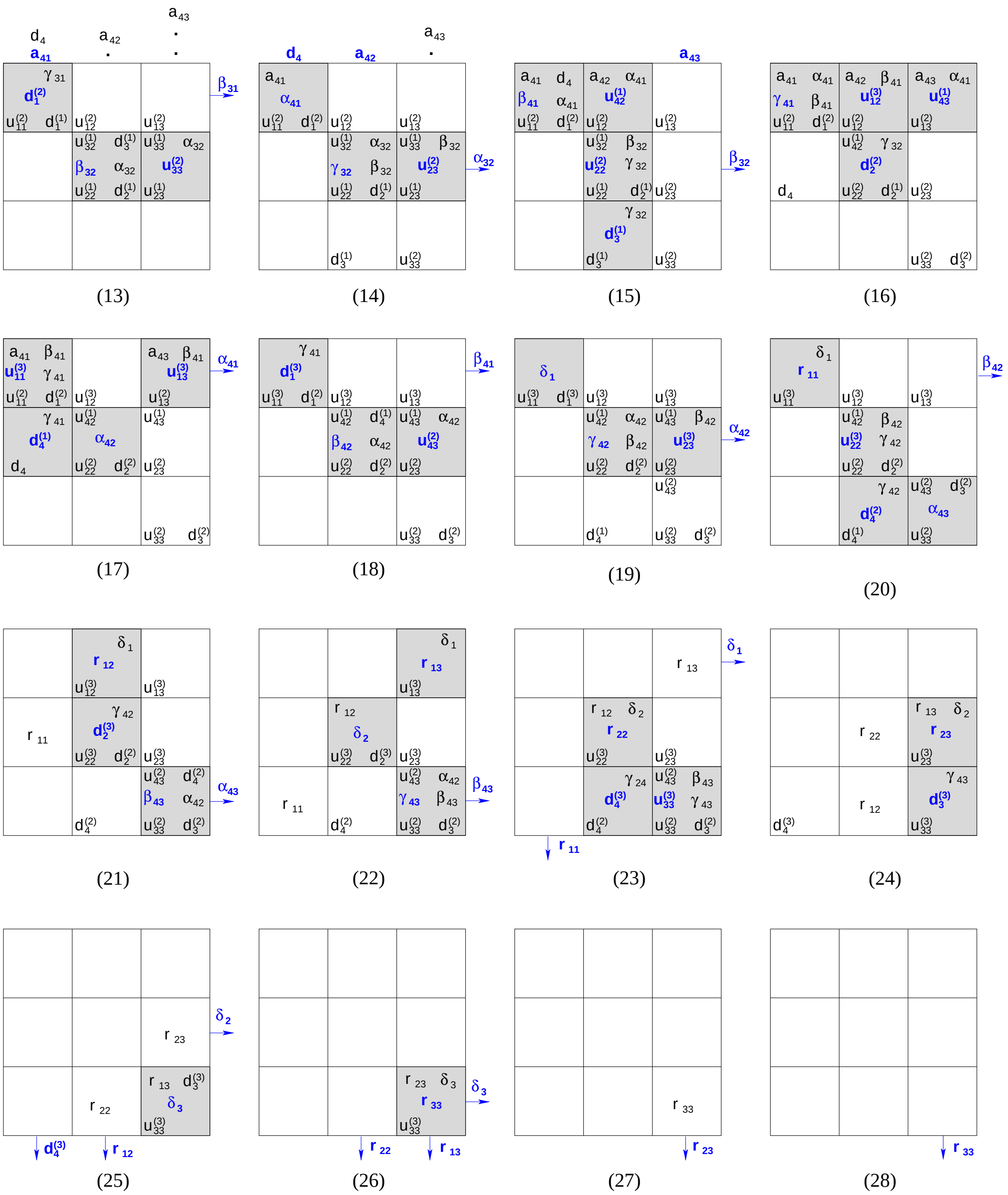

Figure 11: Continuation of Figure 10.

$\gamma_{ij} = -\alpha_{ij}\beta_{ij}$, 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, we compute $\delta_i = 1/\sqrt{d_i^{(n)}}$. The updates of the diagonal elements are scheduled on either the diagonal or subdiagonal processors of the array.

The systolic Givens computation involves a relatively large number of operations even for the small 4×3 example in Figures 10 and 11. Therefore, rather then discussing the computation of a particular element, we describe the data flow through the array 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 10 and 11 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} $(i \leq j)$ via a sequence of intermediate values: a_{ij}, $u_{ij}^{(1)}$, $u_{ij}^{(2)}$, $\ldots$, $u_{ij}^{(n)}$, $r_{ij} = \delta_i u_{ij}^{(n)}$. 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 22 will therefore generate intermediate values of δ_i, that require further updates when computing Equation 24. In Figure 11, 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 16 below.

Figures 12 and 13 illustrate the systolic premultiplication, which updates matrix A according to Equation 23. 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 23. 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 12 and 13 is relatively straightforward. We show the systolic premultiplication of a 4×3 matrix A. According to Equation 23, the premultiplication for this particular example is:

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

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}:

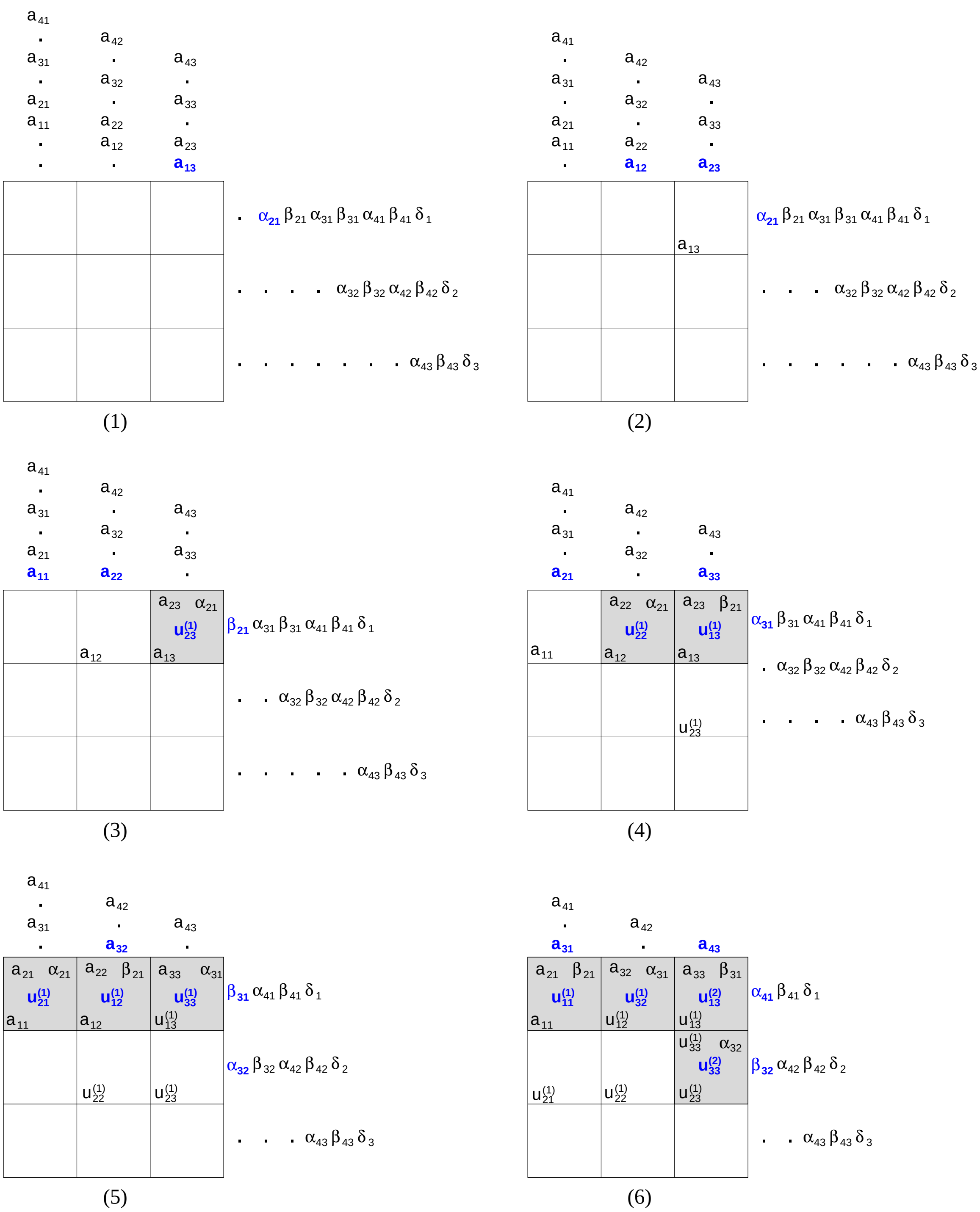

Figure 12: Systolic premultiplication for computing R and update A according to Equation 23. [continued in Figure 13]

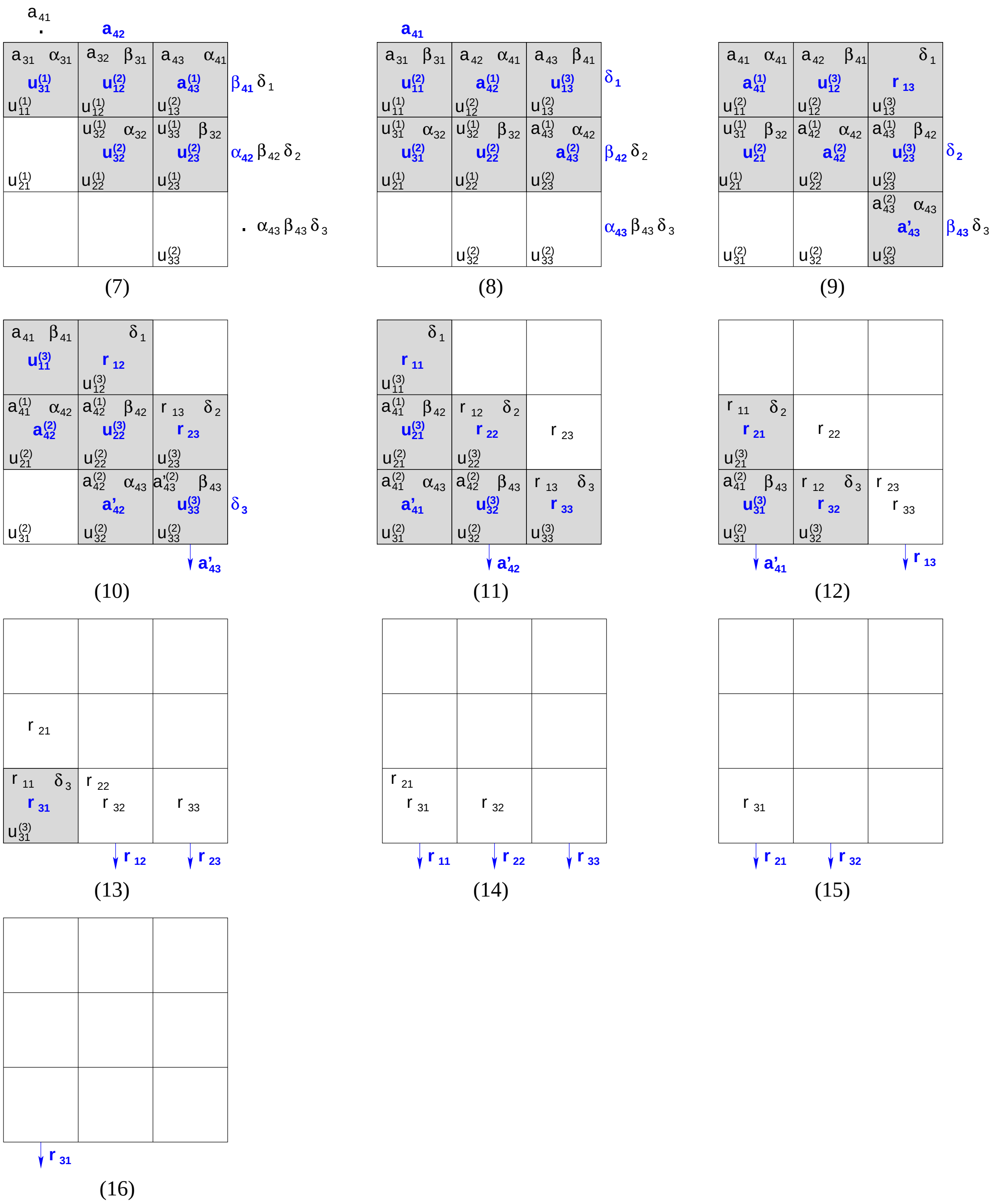

Figure 13: Continuation of Figure 12.

$u_{11}^{(1)} = a_{11} + \beta_{21} \cdot a_{21}$, $u_{11}^{(2)} = u_{11}^{(1)} + \beta_{31} \cdot a_{31}$, $u_{11}^{(3)} = u_{11}^{(2)} + \beta_{41} \cdot a_{41}$, $r_{11} = \delta_1 \cdot u_{11}^{(3)}$. In Figures 12 and 13, 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 14 and 15 illustrate the systolic postmultiplication, which computes matrix Q of the QR factorization according to Equation 27 by postmultiplying the sequence of fast Givens transformations into the identity matrix. Here, we denote intermediate values of G 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 14 and 15 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 15, these are the values $g_{14}^{(3)}$, $g_{24}^{(3)}$, and $g_{34}^{(3)}$. We 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, we leave the updates involving zero-elements in the computation to keep the systolic array as simple as possible.

We are now ready to compose our three systolic algorithms for fast Givens computation, premultiplication, and postmultiplication, and discuss the decoupled data flow of the

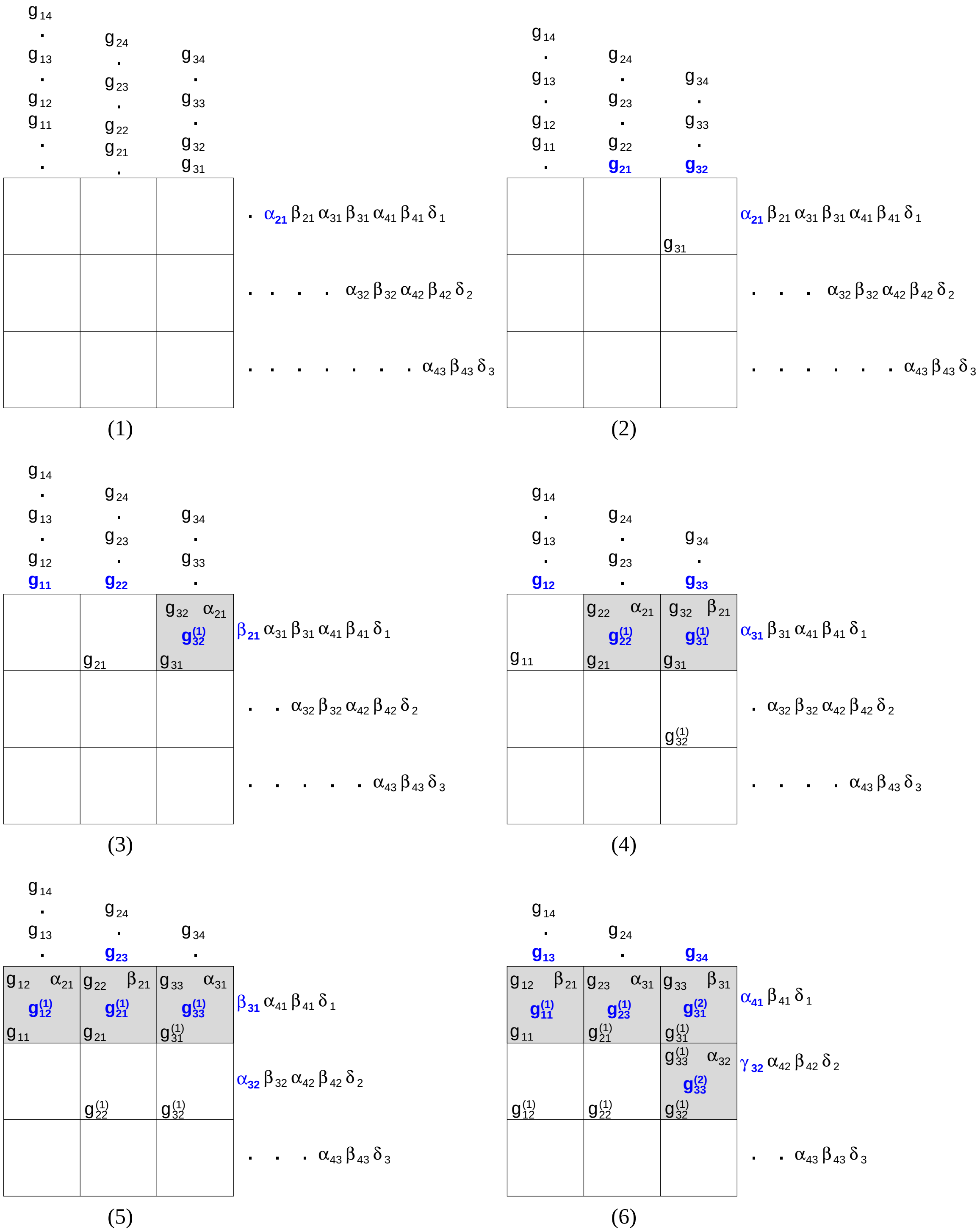

Figure 14: Systolic postmultiplication for computing an $R \times M$ block of $Q = IG(2,1)G(3,1)G(4,1)G(3,2)G(4,2)G(4,3)D^{-1/2}$. [continued in Figure 15]

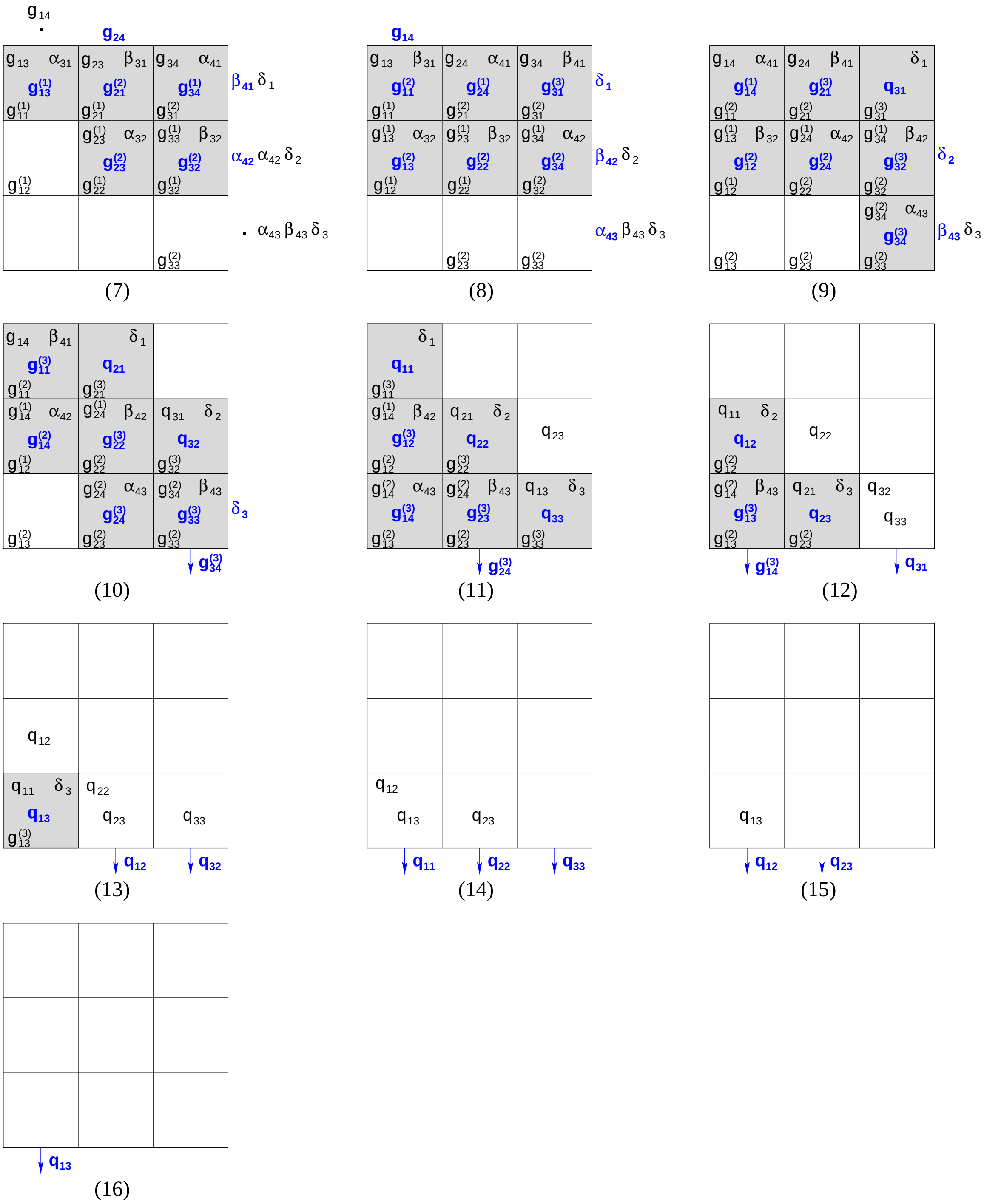

Figure 15: Continuation of Figure 14.

complete QR factorization. Figure 16 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. We use two fast Givens computations to compute R_{11}, R_{22}, and the corresponding fast Givens transformations in phase 1 according to Equation 22 and phase 3 according to Equation 24, respectively. Phase 2 implements the premultiplication according to Equation 23. The remaining phases use the postmultiplication to compute matrix Q. In particular, matrix Q_1 of Equation 25 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 our presentation in Figures 10 and 11. This reflected version of the systolic algorithm allows us to stream the results of the premultiplication from phase 2 right back into the array. Similarly, we use reflected versions of the postmultiplication during phases 7–9. We note that the computation of R is finished after phase 3. Thus, for linear system solvers that do not require knowledge of Q, we could stop the computation after these three phases. Phases 7–12 for the computation of Q deserve further discussion, because our implementation deviates slightly from Equations 25–27. Rather than computing Q_2 explicitly, we 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 26. 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 our other stream algorithms, the partitioning due to Equations 22–26 produces a set of heterogeneous subproblems. We reduce the QR factorization of size $M \times N$ until the subproblems can be computed on an $R \times R$ array of compute processors. Figure 16 represents the data movement when computing the systolic subproblems of Equations 22–27. We use R memory processors at the top of the array to buffer the columns of A and 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, our decoupled systolic QR factorization requires $P = R^2$ compute processors and $M = 3R$ memory processors. We observe that $M = o(P)$ and thus our QR factorization is decoupling efficient.

Efficiency Analysis

In the following, we analyze the efficiency of computing R and Q of an $M \times N$ matrix A separately, because the computation of Q is optional. To handle the general case where $M \geq N$, we introduce two σ-values $\sigma_M = M/R$ and $\sigma_N = N/R$ with network size R. We 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.

We begin by analyzing the number of time steps for the computation of matrix R. We apply the methodology used for the lower-triangular solver and the LU factorization, by

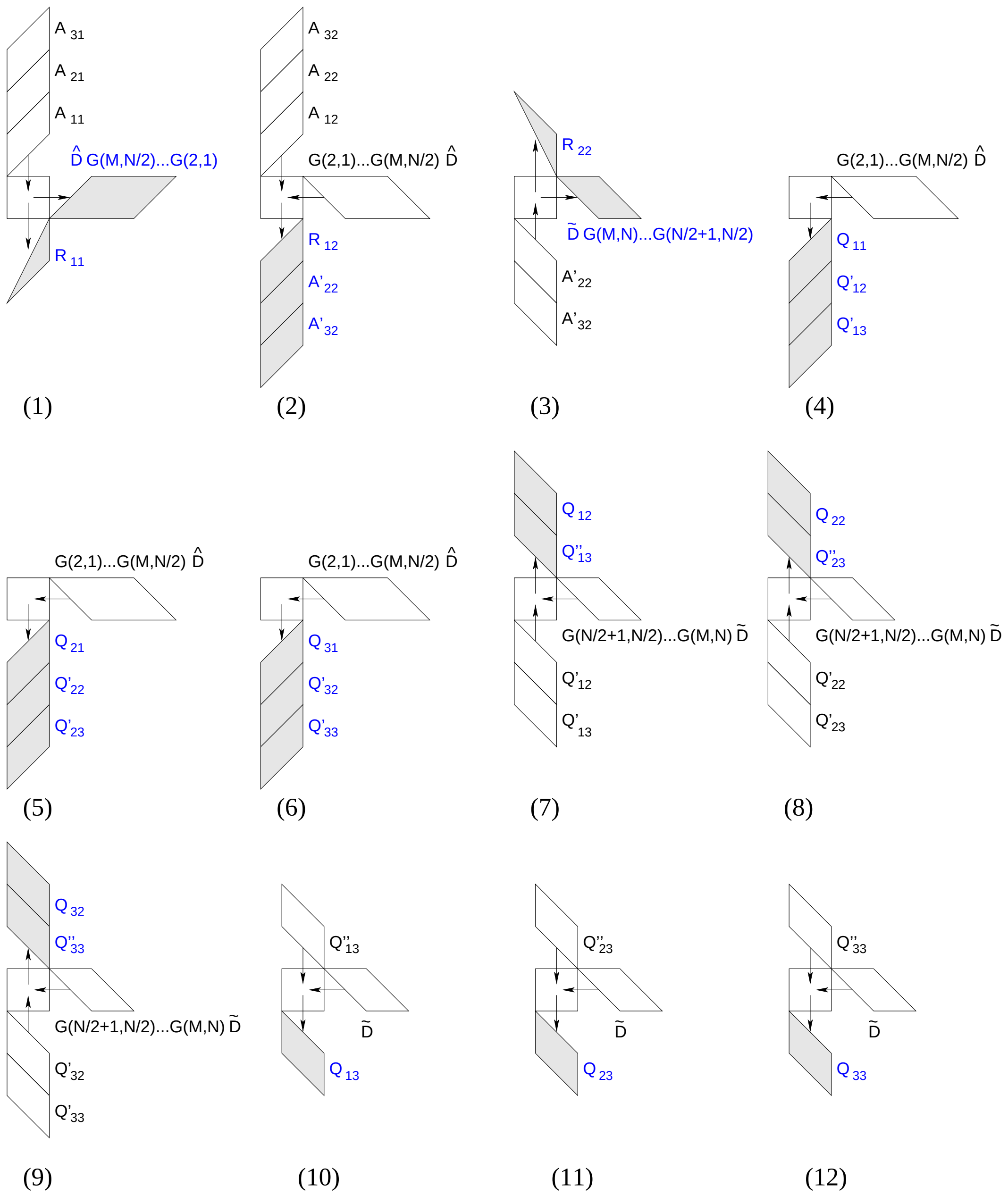

Figure 16: Phases of a stream-structured QR factorization on an $R \times R$ array of compute processors, with $M = 3R$ and $N = 2R$. In phase 1, we triangularize the left-most $N/2$ columns of A, producing R_{11} and a set of fast Givens transformations as expressed in Equation 22. In phase 2, we apply the Givens transformations to the right-most $N/2$ columns of A giving us R_{12}, A'_{22}, and A'_{32} according to Equation 23. In phase 3, we triangularize the right-most columns of A according to Equation 24. Phases 4-6 compute Q_1 according to Equation 25, while phases 7-12 compute $Q = Q_1 Q_2$ according to Equation 27. 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.

partitioning the problem recursively until matrix A consists of $\sigma_M \times \sigma_N$ blocks of size $R \times R$. Equation 28 illustrates the factorization for $\sigma_M = 5$ and $\sigma_N = 4$.

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \\ A_{51} & A_{52} & A_{53} & A_{54} \end{pmatrix} = Q \begin{pmatrix} R_{11} & R_{12} & R_{13} & R_{14} \\ 0 & R_{22} & R_{23} & R_{24} \\ 0 & 0 & R_{33} & R_{34} \\ 0 & 0 & 0 & R_{44} \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{28}$$

To facilitate the efficiency analysis, we discuss our block-iterative schedule in detail. Our partitioning in Equations 22–24 extends to the 5×4 case as follows. The factorization sweeps across pivot blocks from top left to bottom right. For each pivot block, we triangularize and update. First, we annihilate all lower-triangular elements in the pivot column using a systolic fast Givens computation. Then, we update the column blocks to the right of the pivot column by means of premultiplications, cf. phases 1 and 2 in Figure 16. In our example of Equation 28, we first annihilate all lower-triangular elements in column block A_{i1}. Second, we premultiply the 5×3 block matrix consisting of column blocks $(A_{i2}\ A_{i3}\ A_{i4})$ for $1 \leq i \leq 5$. We apply our systolic premultiplication once to each of these column blocks separately. We then proceed 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 17 shows the areas of matrix A that are effected by the fast Givens computation (a) and the premultiplication (b) when handling column block i.

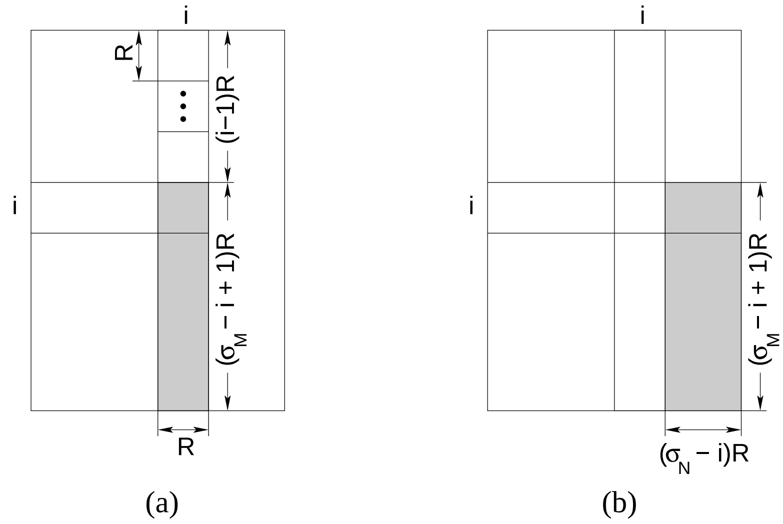

Figure 17: (a) The fast Givens computation of Equation 22 (or Equation 24) for column block i effects the bottom $(\sigma_M - i + 1)R$ rows. (b) After triangularizing column block i, we update the $(\sigma_M - i + 1)R \times (\sigma_N - i)R$ submatrix of A according to Equation 23.

We now 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 10 and 11 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 12 and 13. 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, we have

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

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 $\sigma_M, \sigma_N \gg 1$ we have $E_r(R) \approx R/(R + 3)$, and we achieve more than 90 % efficiency for $R \geq 27$.

We now turn to the efficiency analysis of the computation of matrix Q of the QR factorization. The number of multiply-and-add operations is $C_Q(M, N) = 2M^2N - MN - MN^2$. We use a block-iterative schedule to overlap and pipeline the systolic postmultiplications. We partition the problem 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 18. We apply our 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 14 and 15, 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 of 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 $2R\sigma_M(\sigma_M - i + 1) + 6R$.

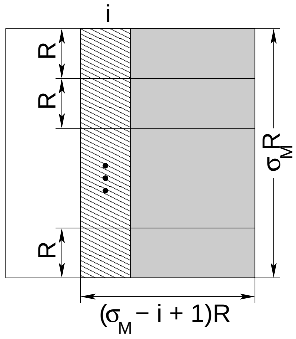

Figure 18: We compute column block i of $M \times M$ matrix Q by partitioning the computation into σ_M row blocks, each with R rows and $(\sigma_M - i + 1)R$ columns. Each row block is computed using a systolic postmultiplication. After the postmultiplication, the hatched area of the matrix holds the final values of Q, while the shaded area comprises intermediate values.

We obtain 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. We 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 matrix approaches the optimal value of 100 %. When $\sigma_M, \sigma_N \gg 1$, we have $E_q(R) \approx R/(R+3)$, and we achieve 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 our 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 we obtain 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 we 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

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

We 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.

8 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, we assume that $M \geq N$. Element k of b is given by

$$b_k = \sum_{i+j=k+1} a_i \cdot w_j \tag{29}$$

where

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

Partitioning

We partition the convolution into N/R subproblems by partitioning the sum in Equation 29 as follows:

$$b_k = \sum_{l=1}^{N/R} \sum_{i+j=k+1} a_i \cdot w_j \tag{30}$$

where

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

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

Decoupling

We use the systolic design of Figure 19 to implement a convolution with $N = R$. This design is independent of the length M of vector a. For the example in Figure 19 we have chosen $N = R = 4$ 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. In contrast, vector a streams 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.

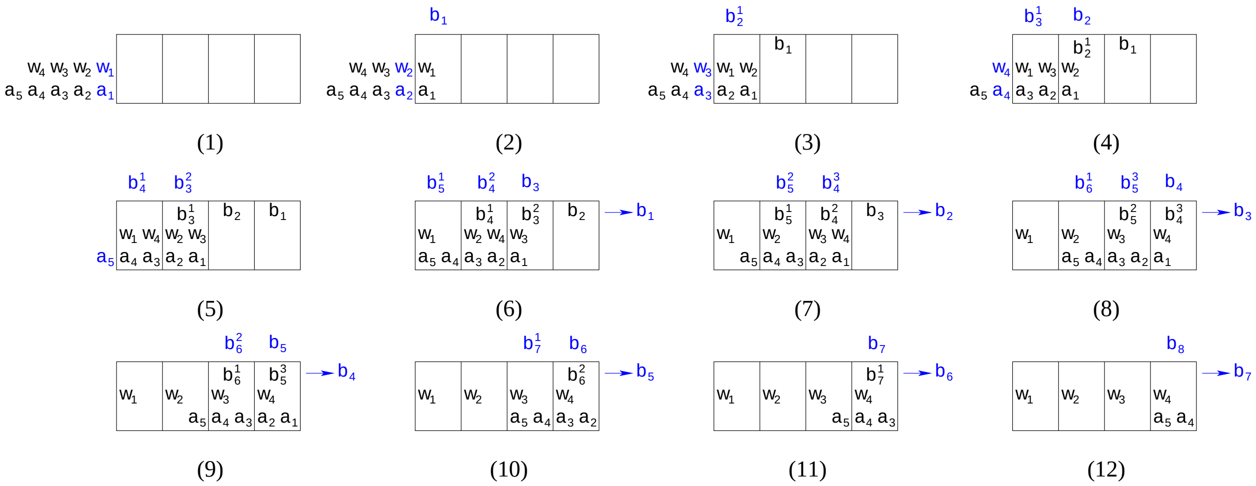

Figure 19: Systolic convolution of input sequence a_i of length $M = 5$ with $N = 4$ weights w_j. Both the weights and input sequence are fed into the linear array of $R = 4$ compute processors. Intermediate results are shown above the corresponding processors. Value b_k^i represents an intermediate value of b_k after the first i products have been computed according to Equation 29.

We illustrate the data movement in Figure 19 by discussing the computation of $b_4 = a_4 w_1 + a_3 w_2 + a_2 w_3 + a_1 w_4$. We begin with time step 5 in Figure 19 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 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.

We use the partitioning of Equation 30 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, we 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 20 illustrates the computation of a convolution on a linear processor array. Our decoupled systolic convolution requires $P = R$ compute processors and $M = 2$ memory processors. We observe that $M = o(P)$ and, therefore, our convolution is decoupling efficient.

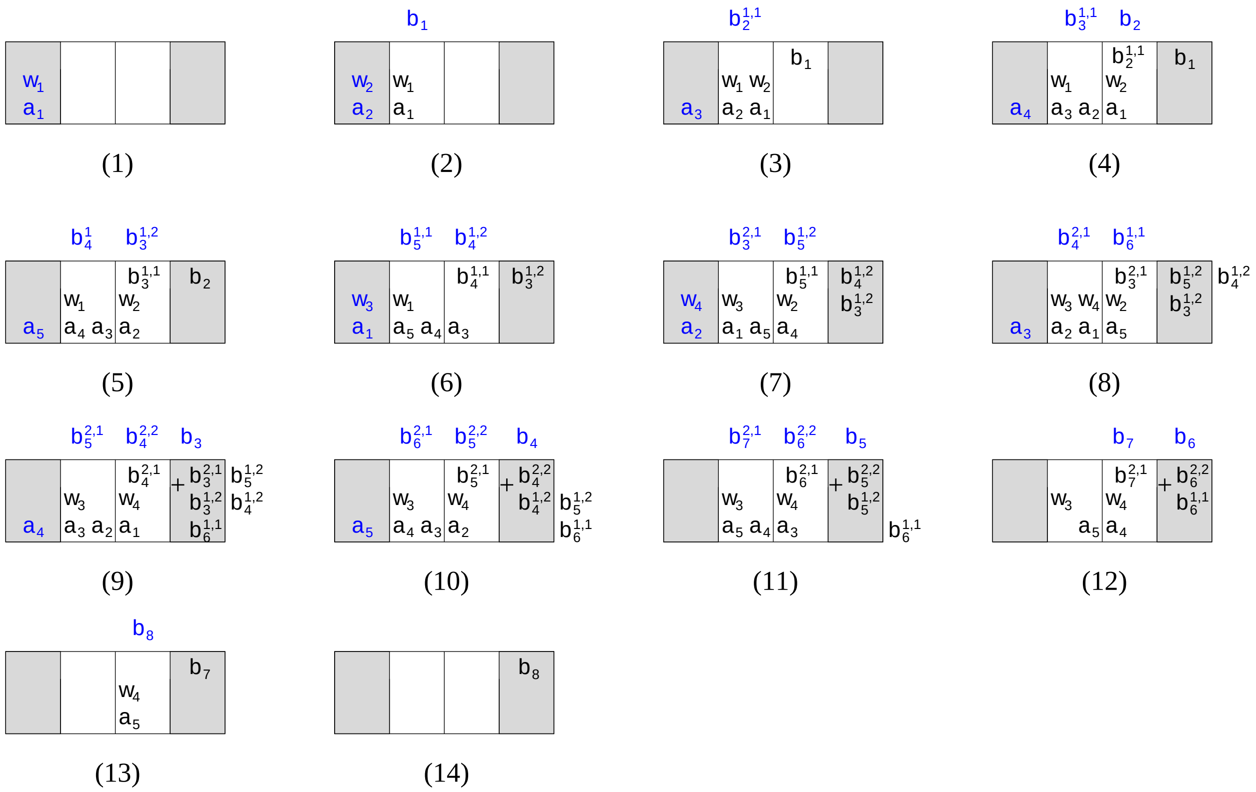

Figure 20: Stream convolution of an input sequence of length $M = 5$ with $N = 4$ weights on a linear array of $R = N/2 = 2$ compute processors and $M = 2$ memory processors. Value $b_k^{l,i}$ represents the computation of b_k when the outer summation of Equation 30 has been executed l times and the 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.

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, we partition 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 20.

We account for the time steps of the convolution as follows. There are $\sigma = N/R$ systolic convolutions on a linear array of R compute processors that 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, and because each processor executes one multiply-and-add operation per time step. Due to the perfect overlap of subsequent systolic convolutions, the R time steps needed to drain the pipeline are incurred only once. Thus, the number of time steps is:

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

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:

$$E_{conv}(\sigma, R) \quad = \quad \frac{\sigma^2}{\sigma^2 + N/M} \cdot \frac{R}{R+2} \tag{31}$$

Given our assumption that $M \geq N$, we have $N/M \leq 1$, and the efficiency of our stream-structured convolution approaches the optimal value of 100 % for large values of σ and R. Thus, for $\sigma \gg 1$, we have $E_{conv} \approx R/(R + 2)$ and obtain 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

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

We note that for $N = M$ the efficiency of the systolic convolution has an upper bound of 50 %. Our stream convolution defies this upper bound provided that σ is sufficiently large.

9 Conclusion

We have studied the feasibility of highly efficient computation on tiled architectures. Because these architectures are constrained by short wires, we found that systolic computation, which considers both architecture and algorithms simultaneously, proves invaluable even with today's microtechnology. We developed stream algorithms together with a decoupled systolic architecture as a means to exploit the similarity between systolic arrays and tiled architectures. In addition, we discovered that for many regular applications decoupling memory accesses from computation allows us to move load and store operations off the critical path. The resulting reduction of the critical-path length enables us to increase efficiency by increasing the number of processors, and even approach 100 % compute efficiency in the limit.

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

Application	$P(R)$	$M(R)$	S	$T(\sigma, R)$	$E(\sigma, R)$
matrix mult.	R^2	$2R$	9	$\sigma^3 R + 3R$	$\frac{\sigma^3}{\sigma^3+3} \cdot \frac{R}{R+2}$
triangular solver	R^2	$3R$	9	$\frac{R}{2}(\sigma^3 + \sigma^2 + 6\sigma - 2)$	$\frac{\sigma^3}{\sigma^3+\sigma^2+6\sigma-2} \cdot \frac{R}{R+3}$
LU factorization	R^2	$3R$	9	$R(\frac{1}{3}\sigma^3 + \frac{1}{2}\sigma^2 + \frac{31}{6}\sigma - 2)$	$\frac{\sigma^3}{\sigma^3+\frac{3}{2}\sigma^2+\frac{31}{2}\sigma-6} \cdot \frac{R}{R+3}$
QR factorization	R^2	$3R$	22	$R(\frac{5}{3}\sigma^3 + \frac{7}{2}\sigma^2 + \frac{125}{6}\sigma - 3)$	$\frac{\sigma^3}{\sigma^3+\frac{21}{10}\sigma^2+\frac{25}{2}\sigma-\frac{9}{5}} \cdot \frac{R}{R+3}$
convolution	R	2	11	$\sigma M + R$	$\frac{\sigma^2}{\sigma^2+N/M} \cdot \frac{R}{R+2}$

Table 1: Summary of stream algorithms. We show the number of compute processors P, number of memory processors M, and the bounded amount of storage for local variables per compute processor S. In addition, we compare the execution time T and compute efficiency E.

We presented five concrete examples of stream algorithms for a matrix multiplication, a triangular solver, an LU factorization, a QR factorization, and a convolution. Table 1 summarizes our results for these applications. For each of our stream algorithms, the table lists the number of compute processors P and memory processors M as a function of network size R. We also show the amount of local state S as the number of scalar variables that each compute processor must maintain. Consider the matrix multiplication, for example. Each compute processor must store locally the values of problem size N, network size R, its row and column indices within the compute array, the intermediate value of the matrix element being computed, a computed product element that is streamed towards the output, and three loop variables. Note that the space requirements for the local state on the compute processors is bounded for each of our stream algorithms by a reasonably small number. Table 1 also compares the execution times T and efficiencies E of our stream algorithms.

Our experience with the design of stream algorithms has revealed three noteworthy insights. First of all, our design philosophy for stream algorithms appears to be quite versatile. We were able to formulate stream algorithms even for relatively complex algorithms like the QR factorization. Secondly, our stream algorithms achieve 100 % compute efficiency where conventional systolic designs cannot. The astute reader may have noticed that none of our systolic algorithms achieves 100 % efficiency. For an infinitely large network size R, the efficiency of our systolic matrix multiplication is $E_{mm} = 1/3$, for the triangular solver $E_{lts} = 1/6$, for the LU factorization $E_{lu} = 1/12$, for the QR factorization $E_{qr} = 5/69$, and for the convolution $E_{conv} = 1/(1 + N/M)$. However, all of our stream algorithms are compute efficient. Thirdly, our design philosophy for stream algorithms emphasizes the amortization of inefficient systolic computations by means of efficient ones. For example, in the QR factorization, the systolic algorithm for the Givens computation is quite inefficient, because only the diagonal processors of the compute array are fully utilized. However, the number of multiply-and-add operations required by the QR factorization is not dominated by the systolic Givens computation, but rather by the systolic premultiplication and postmultiplication, both of which are highly efficient, because we can pipeline a large number of problems. When partitioning a large problem into systolic subproblems, we focus our efforts on identifying and optimizing those critical subproblems which dominate the computational effort.

In summary, we believe that stream algorithms provide an excellent match for the physical and technological constraints of future single-chip tiled microarchitectures.

Acknowledgments

This work has been funded as part of the Raw project by DARPA, NSF, the Oxygen Alliance, MIT Lincoln Laboratory and the Lincoln Scholars Program. We would like to thank Janice McMahon and Bob Bond for their support.

References

[1] 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.

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

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

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

[5] 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.

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

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

[8] 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.

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

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

[11] 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.

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

[13] 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.

[14] 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.

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