Journal of Instruction-Level Parallelism 6 (2004) 1-31 Submitted 6/04; published 9/04

Stream Algorithms and Architecture

Volker Strumpen STRUMPEN@CSAIL.MIT.EDU
Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Stata Center, 32 Vassar Street, Cambridge, MA 02139
Henry Hoffmann HANK@MIT.EDU
Lincoln Laboratory, Massachusetts Institute of Technology, 244 Wood Street, Lexington, MA 02420
Anant Agarwal AGARWAL@CSAIL.MIT.EDU
Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Stata Center, 32 Vassar Street, Cambridge, MA 02139

Abstract

Wire delay and power consumption are primary obstacles to the continued scaling of microprocessor performance. Fundamentally, both issues are addressed by the emerging breed of single-chip, tiled microarchitectures including Raw [1], Trips [2], Scale [3], Wavescalar [4], and Synchroscalar [5], that replicate programmable processing elements with small amounts of memory and communicate via on-chip networks characterized by extremely low latencies and high bandwidth. Our goal is to show that tiled microarchitectures permit energy-efficient high-performance computations when algorithms are mapped properly. To that end, we propose a *decoupled systolic architecture* as a canonical tiled microarchitecture that supports the resource requirements of *decoupled systolic algorithms* designed specifically for this architecture. We develop an analytical framework for reasoning about the efficiency and performance of decoupled systolic algorithms. In particular, we define *stream algorithms* as a class of decoupled systolic algorithms that achieve $100\,\%$ computational efficiency asymptotically for large numbers of processors. We focus our attention on the class of regularly structured computations that form the foundation of scientific computing and digital signal processing.

1. Introduction

As we approach the physical limits of microtechnology, signal propagation delays and power consumption emerge as the primary focus of engineering trade-offs. Tiled microarchitectures cope with large signal propagation delays across large chip areas by using short wires and exposing the scheduling of wire resources to software [1]. While tiling provides a path to increasing clock frequency as a primary guarantor of performance growth, power consumption is becoming the dominant constraint. The continuation of established VLSI design practices would produce silicon chips that generate more heat than packaging technology can dissipate. In this situation, energy efficiency is becoming a primary design aspect for microarchitectures. Our approach is to increase energy efficiency by boosting computational efficiency. The key lies in finding a good match of algorithms and microarchitecture. In this paper, we show how we can achieve maximal computational efficiency for regular applications on a programmable tiled microarchitecture.

Energy efficiency is a direct consequence of hardware utilization measured in operations per time unit [6]. We note that applications that utilize the underlying hardware resources to $100\,\%$ with useful computation are not only the highest performing but also the most energy efficient. Hard-

©2004 AI Access Foundation and Morgan Kaufmann Publishers. All rights reserved.

ware utilization depends on the statistical distribution of executed instruction types, the instruction schedule, and, of course, the data which determine how often voltages toggle on data wires. Previous studies have shown that in RISC processors [7] or in tiled microarchitectures like Raw [8] at most $50\,\%$ of the energy is consumed by the functional units performing useful work. The majority of the energy is dissipated by the clock network and on-chip memories.

To show that tiled microarchitectures can perform energy-efficient computations, we propose a ***decoupled systolic architecture*** (***DSA***) as a canonical tiled microarchitecture for executing ***decoupled systolic algorithms***. A DSA consists of an array of compute tiles, peripheral memory tiles, and an off-chip memory system. As the name suggests, decoupled systolic algorithms decouple memory accesses from computation [9] and execute in a systolic fashion [10]. We define ***stream algorithms*** as the class of decoupled systolic algorithms that achieve $100\,\%$ computational efficiency on the DSA asymptotically for large numbers of processors. For comparison, many existing systolic algorithms miss this efficiency target [10, 11, 12, 13]. The crux behind stream algorithms is a reduction of power consumption by memory modules so as to be asymptotically insignificant compared to the power consumed by useful computation. This article makes the following contributions:

1. We demonstrate by construction that there exist formulations of many regular algorithms as stream algorithms, which by definition achieve $100\,\%$ computational efficiency asymptotically for large numbers of processors on a DSA.
2. We present a design methodology to guide the design of stream algorithms as a well defined entity. The resulting rigorous approach improves our understanding of tiled microarchitectures as a microarchitectural design point, enables a systematic restructuring of algorithms, and communicates our perspective on the interaction between algorithms and architecture.
3. Our efficiency analysis allows us to identify machines as decoupled systolic architectures, by explicitly incorporating the area cost of per-tile local memories, and penalizing the use of unbounded local memories. This is particularly important in the context of emerging single-chip tiled microarchitectures which use individual tiles with small silicon footprints.
4. We have identified the DSA as a canonical tiled microarchitecture, which resembles systolic machines for streaming applications and achieves $100\,\%$ asymptotic energy efficiency on an important class of algorithms. We believe our insight transforms a well-known constraint of tiled microarchitectures, small local memories, into a feature related to energy efficiency.
5. Finally, we present experimental results of executing several stream algorithms on an existing tiled microarchitecture by emulating the decoupled systolic architecture. These results indicate that stream algorithms are not just of theoretical interest, but that they can achieve high performance and energy efficiency for practical system sizes.

Throughout this paper, computational efficiency shall denote the utilization of floating-point units, because we study algorithms dominated by floating-point arithmetic. To enable simple counting arguments for algorithmic design and efficiency analysis, we distinguish between computation and memory accesses only, and do not account for any other contributors including the clock network. This choice is based on the fact that computation and memory accesses are the first-order effects that determine computational and energy efficiency within the realm of algorithmic analysis. Viewing energy efficiency as a consequence of computational efficiency, the algorithm designer can focus on the goal of maximizing the latter.

The remainder of this paper is organized as follows. In Section 2 we present the decoupled systolic architecture. Section 3 defines our notion of stream algorithms. Then, we discuss the algorithmic design process of transforming an application into a stream algorithm using three sample applications; more stream algorithms can be found in [14, 15]. We begin with a matrix multiplication in Section 4, which is a fundamental stream algorithm and a workhorse used as part of other stream algorithms. Then, we present two less obvious stream algorithms, a triangular solver in Section 5 and a convolution in Section 6. We analyze each stream algorithm, and argue why it achieves optimal computational efficiency of $100\,\%$ asymptotically when executed on our decoupled systolic architecture. We discuss related work in Section 7, summarize our results including emulation experiments on Raw in Section 8, and conclude in Section 9.

2. The Decoupled Systolic Architecture

Our decoupled systolic architecture (DSA) is a ***canonical tiled microarchitecture*** consisting of an array of processing elements replicated across the silicon area and connected by an on-chip network. Tiled microarchitectures [1] are single-chip parallel machines whose organization is primarily determined by the propagation delay of signals across wires [16]. To enable high clock frequencies on large chip areas, tiled microarchitectures have short wires that span a fraction of the side length of a chip, and use registers to pipeline 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 [17], parallel computation [13], parallelizing compiler design [18], and instruction-level parallelism [19] 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 [10, 20, 21], 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 [22] became programmable, so they could reap the benefits of systolic arrays for regular applications. 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.

Our decoupled systolic architecture serves as a canonical tiled microarchitecture. It is the result of the design and analysis of various foundational decoupled systolic algorithms. To execute decoupled systolic algorithms efficiently, each tile needs no more than a small, bounded amount of local memory; an algorithmic feature exploited by the DSA. Tiles are arranged as an array with a set of memory tiles on the periphery as illustrated in Figure 1. The array consists of $R \times R$ ***computation tiles***, denoted P, and $4R$ ***memory tiles***, denoted M, on the periphery. We use letter R to denote the ***network size*** of the array analogous to the *problem size* N of an algorithm.[1]

A decoupled systolic 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

1. We use the *network size R* as a canonical network parameter. The number of tiles of a network of size R is determined by the network topology. For example, a linear array of size R contains R computation tiles and 2 memory tiles, whereas a 2-dimensional array contains R^2 computation tiles and $4R$ memory tiles. Our DSA is a 2-dimensional, square array of side length R.

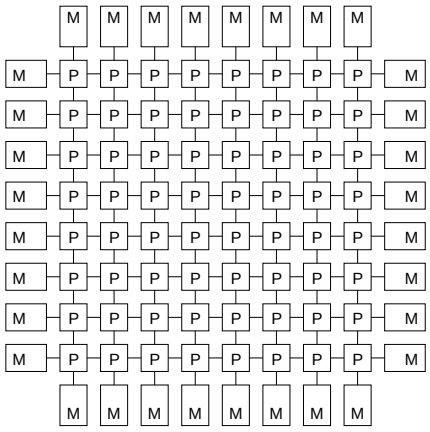


Figure 1: A decoupled systolic architecture (DSA) is an $R \times R$ array of computation tiles (P) surrounded by $4R$ memory tiles (M), shown for network size $R = 8$.

data are supplied to the computation tiles from the memories in a continuous stream via the network, thereby eliminating load/store memory accesses on the computation tiles. As a consequence, decoupled systolic algorithms decouple memory accesses from computation [9], and move the memory accesses off the critical path. It is the decoupling of memory accesses from computation that permits an energy efficient microarchitecture such that the majority of the power is consumed by computation rather than memory accesses.

2.1 Tile Architecture

The decoupled systolic architecture has two types of tiles: *computation tiles* and *memory tiles.* The peripheral memory tiles consist of a computation processor augmented with additional local memory, backed by a large off-chip memory system. As the name suggests their primary use involves memory accesses including address computations. We assume that the memory can deliver a throughput of two loads and one store per clock cycle. This section focuses on the key architectural features of the computation tiles.

The computation tile, sketched in Figure 2, is a simple general-purpose programmable core comprising an integer unit, a floating-point unit with a multiply-and-add module as the centerpiece, and a multi-ported, general-purpose register file. Our analysis of decoupled systolic algorithms has revealed that 64 registers suffice. To focus our attention on the datapath, Figure 2 omits all of the control logic and a small instruction buffer. We assume a single-issue, in-order pipelined FPU that allows us to issue one multiply-and-add operation per clock cycle. Other functional units, including a divider and a square root unit, should be pipelined, although they are less critical for application performance due to less frequent use. When referring to ***computational efficiency***, we typically mean the utilization of the FPU. In particular, a computational efficiency of $100\,\%$ refers to the FPU executing one operation per clock cycle.

Arguably the most important feature of the DSA is the design of its on-chip network. Our interconnect uses a ***register-mapped network*** [1, 22, 23, 24, 25], that allows instructions to access the network via register names. Unlike Raw [1], the DSA does not incorporate a separate switch proces-

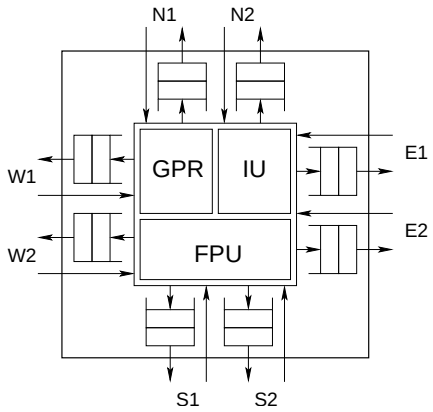


Figure 2: A computation tile contains a general-purpose register file (GPR), an integer unit (IU), and a floating-point unit (FPU) based on a multiply-and-add module. The processor connects via FIFO's to its four neighbors.

sor for programmed routing. The flexibility provided by a separate switch processor, in particular for routes bypassing the main processor, is simply not needed for stream algorithms. However, DSA's can be emulated on top of a tiled microarchitecture like Raw, although with a modest loss in efficiency as discussed in Section 8.

As illustrated in Figure 2, we use FIFO's to connect and synchronize neighboring processors. The implementation of stream algorithms demands the existence of two bidirectional connections between each of the four neighboring tiles. The FIFO's are blocking and are exposed to the programmer by mapping them into register names of 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. Like Raw, 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. The tight network integration ensures a latency of a single clock cycle for communication between neighboring computation tiles, and allows for efficient pipelining of results from operations with different pipeline depths through the processor array.

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

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

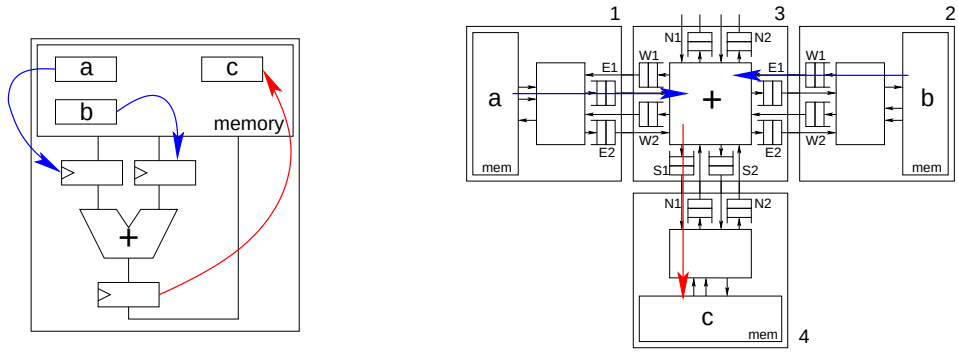
consists of two parts. The `fma` operation is a floating-point multiply-and-add compound instruction. It multiplies the values arriving on network ports `N1` and `W2`, and adds the product to the value in general-purpose register `$4`. Simultaneously, it routes the incoming values to the neighboring tiles 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 DSA block until all operands are available. Using small FIFO's as deep as the deepest functional unit eases the problem of scheduling instructions substantially. There exists a trade-off between instruction width and area

occupied by the corresponding wires within a tile. Based on our analysis of stream algorithms, we assume that up to three data movements can be specified within the route part of an instruction.

2.2 On-chip Network

Our discussion of the register-mapped network in the previous section nearly completes the description of the on-chip network. In fact, the on-chip network consists of little more than the wires connecting neighboring tiles, and a switch within each tile with programmed connections between the wires themselves as well as the functional units. Since the wires span a distance no longer than the side length of a tile, they can be relatively short and fast. A good VLSI implementation balances the side length of a tile with the critical-path length within a tile. In the following, we provide a brief introduction to the use of this network organization by means of a programming example.

From the programmer's point of view, the essential features of a register-mapped network are that it enables us (1) to distribute a computation across multiple tiles, (2) to use the network as temporary storage, and (3) to reduce the critical path length of the computation by abandoning local memory accesses of conventional load/store instructions. As a simple example illustrating the decoupling of a computation, consider the computation of the sum $c = a + b$. Figure 3 shows a version of the addition on a conventional load/store processor architecture on the left-and side, and a space-distributed 4-tile version for the DSA with network size $R = 1$ on the right-hand side.



(a) Load/Store Architecture (b) Decoupled Systolic Architecture

Figure 3: Dataflows of $c = a + b$ on a load/store architecture (a) and on four tiles of a DSA (b).

On a conventional load/store processor the addition requires four instructions to load operands `a` and `b` from memory into general-purpose registers, perform the addition, and store the result in variable `c` (we use the $\&$-symbol to indicate that `a`, `b`, and `c` denote memory locations):

```
lw   $1,&a
lw   $2,&b
add  $3,$1,$2
sw   $3,&c
```

In the version for the DSA, we distribute these operations across four tiles, one computation tile and three memory tiles. We execute one instruction per tile, and use the network as temporary storage:

```
      Memory tile 1:  lw   $E1,&a
      Memory tile 2:  lw   $W1,&b
Computation tile 3:   add  $S1,$W1,$E1
      Memory tile 4:  sw   $N1,&c
```

As introduced in Section 2.1, `$E1`, `$W1`, `$N1` and `$S1` are the names of network-mapped registers, which grant read or write access to the networks depending on their use in the instruction. Memory tile 1 loads operand `a` from its local memory and writes it into network register `$E1`. Similarly, memory tile 2 loads operand `b` straight into the network via register `$W1`. Computation tile 3 uses one instruction to read both operands from the network via registers `$W1` and `$E1`, and to write the result back into network register `$S1`. Finally, memory tile 4 reads the sum from network register `$N1` and stores it in its local memory. Since network register names refer to directions, the code is position independent of the particular tile. Note that the register-mapped network acts like a distributed register file, organized as several FIFO's. Furthermore, while the load/store version of the addition occupies four instruction slots on one tile, the 4-tile version occupies one instruction slot on each of the four tiles. Thus, the critical-path length of the program in terms of instruction slots is four times smaller in the 4-tile version.

Let us now consider the element-wise addition of two vectors rather than scalar numbers. In this case, we replicate a loop over array index `i` across the four tiles:

```
1: for (i=0; i<N; i++)    3: for (i=0; i<N; i++)    2: for (i=0; i<N; i++)
     lw  $E1,a[i]              add $S1,$W1,$E1           lw  $W1,b[i]

                          4: for (i=0; i<N; i++)
                               sw  $N1,c[i]
```

With proper loop unrolling, this code achieves a throughput of one addition per clock cycle. In contrast, the equivalent loop of the load/store version achieves a throughput of one addition per four clock cycles only. Thus, investing four times the space improves the throughput by a factor of four. The computational efficiency of both versions is only $25\,\%$, however. In the load/store version, only every fourth instruction is a useful addition, and in the 4-tile version, only one tile out of four, namely computation tile 3, operates at $100\,\%$ computational efficiency, while the three memory tiles implement the data movement. We will see in Sections 4–6 how the design of stream algorithms allows us to utilize all computation tiles of the DSA at $100\,\%$ computational efficiency.

2.3 Off-chip Network

A single-chip DSA will be embedded in a larger system. At the least, we desire access to a larger off-chip memory system. In addition, for the sake of scalability, we want to interconnect multiple chips to assemble larger stream fabrics. For a DSA with a significant portion of the silicon real estate invested in the on-chip network, seamless connectivity to the memory or other DSA chips is essential for scalability and programmability. However, the number of pins provided by chip packaging technology imposes a serious limitation on the number of network wires that may cross the chip boundary. If we were to extend our on-chip network across the chip boundary, we would need an enormous number of data pins. An $R \times R$ array has 4 sides, each with R (memory) tiles, 2 networks with 2 directions and B bits each. Multiplying all factors together results in $16RB$ as

the minimum number of data pins for a single chip. To be concrete, for $R = 8$, a word size of $B = 32$ bits would require 4,096 data pins and a word size of $B = 64$ bits would require 8,192 data pins. Such a number of pins is not only technologically infeasible for the foreseeable future but also area inefficient.

Rather than compromising the network size R to match the availability of pins, a better alternative is to invest in an off-chip network that reduces the number of wires needed to cross the chip boundary. A prime candidate for such a network is a fat-tree [26, 27], that permits concentration in the switches, so that the number of wires connected to the switch at the root of the tree is small compared to the total number of wires connected to the switches at the leaves. Figure 4 shows a fat-tree network arranged such that it forms a back-plane for the memory tiles. The reduction of wires in the switches of a fat tree is quite flexible, allowing us to pick the number of wires X that cross the chip boundary according to the constraints of the packaging technology.

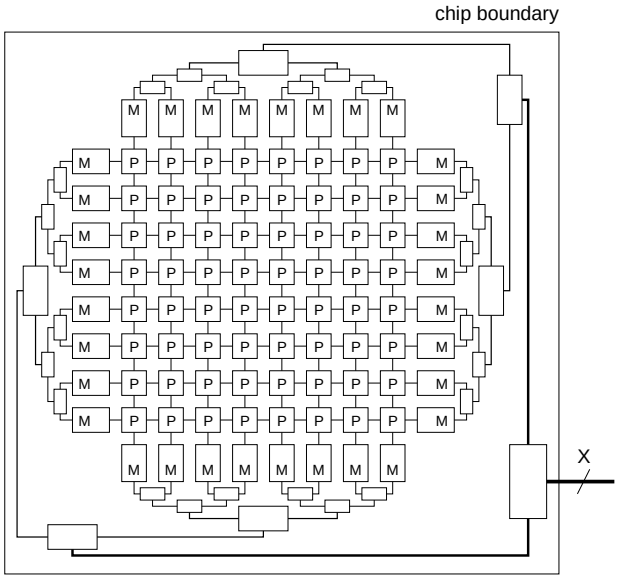


Figure 4: The memory tiles connect via a back plane, a fat-tree network, to the off-chip world, including a memory system. The number of data pins of the chip package is X.

This raises the question of how many wires X we actually wish to have in a single-chip DSA with a fat-tree network as back-plane. Assume that the fat tree connects the chip to a large pool of off-chip memory banks with the memory tiles acting as stream buffers [28] or stream caches [29]. Such an organization of a memory hierarchy should allow us to stream data in and out of the chip at the speed required to ensure $100\,\%$ computational efficiency. To meet this requirement, we need to focus our attention on the bandwidth of the off-chip network.

Our analysis of stream algorithms has revealed that the bandwidth needed across the chip boundary is a factor of four smaller than the bandwidth offered by extending the on-chip network. In fact, our stream algorithms with the largest bandwidth requirements, the stream SVD [15] for example, require two sets of incoming and two sets of outgoing streams through the four sides of our $R \times R$ array. This corresponds to utilizing just 2 sides of R tiles each with only 1 bidirectional network

per tile, leading to an aggregate memory bandwidth of $4RB$. Thus, with $R = 8$ and a word size of 32 bits, the fat-tree network requires only 1,024 data pins. Such a chip is realizable with today's packaging technology [1].

As an aside, it would be desirable to implement a separate on-chip network connecting each computation tile with at least one of the peripheral memory tiles so as to offer a reasonably simple means for supplying the computation tiles via the off-chip network with instructions.

3. Stream Algorithms

In this section we introduce stream algorithms and a set of conditions which enable us to increase efficiency by increasing the number of tiles such that the computational efficiency approaches $100\,\%$ asymptotically, while the power consumed by memory operations becomes insignificant.

The key strategy for the design of stream algorithms is to recognize that the number of memory tiles must be negligible compared to the number of computation tiles because memory tiles do not contribute any useful computation, but tend to consume a relatively large amount of energy. In the following, we call a systolic algorithm whose memory accesses are executed on a different tile than the computation a ***decoupled systolic algorithm*** in recognition of the *decoupled access/execute architecture* [9].[2] While it is often impossible to design an efficient decoupled systolic algorithm for a small number of tiles and a small problem size, we can actually increase the efficiency for larger numbers of tiles and large problems. We emphasize this observation by formulating the decoupling-efficiency condition.

Definition 1 ***(Decoupling-Efficiency Condition)***
Given a decoupled systolic algorithm for a network of size R, let $P(R)$ be the number of computation tiles and $M(R)$ the number of memory tiles. 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 tiles becomes insignificant relative to the number of computation tiles as we increase the network size R. Decoupling efficiency is a necessary condition to amortize the lack of useful computation performed by the memory tiles and the power invested in memory accesses. For example, suppose we implement an algorithm on $P = R^2$ computation tiles. If we can arrange the memory tiles such that their number M 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 $O(\lg R)$, or $O(\sqrt{R})$, or $M = O(R)$, or $M = O(R \lg R)$, for example. In contrast, a design with $M = \Omega(R^2)$ would not be efficiently decoupled. Decoupled algorithms per se are independent of a particular architecture. Note, however, that the DSA is particularly well suited for executing either one such algorithm with $\langle P, M\rangle = \langle \Theta(R^2), \Theta(R)\rangle$ or multiple algorithms concurrently with $\langle P, M\rangle = \langle \Theta(R), \Theta(1)\rangle$.

Decoupling efficiency is a necessary but not sufficient condition to guarantee high efficiency. In order to formulate a sufficient condition, we first clarify our notion of computational efficiency by means of the following definition.

2. Although *systolic algorithms* were originally designed for *systolic arrays* of hardware circuits including registers rather than large local memories with a load/store interface, they have taken on a broader meaning over time to include accesses to potentially large local memories [11, 21].

Definition 2 ***(Computational Efficiency)***
Given an algorithm with problem size N with a number of useful computational operations $C(N)$, a network of size R, the execution time given as number of time steps $T(N,R)$, and the area counted in number of tiles $P(R)+M(R)$, then the ***computational efficiency*** *is*

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

The numerator of Equation 1 consists of the number of useful operations, and the product in the denominator can be interpreted as the computational capacity of the DSA during time period T. For additional insight, we derive the definition of computational efficiency from the familiar notion of speedup, and interpret the denominator of Equation 1 as the space-time product AT of the VLSI model of computation [30]. To that end, we define the area (i.e. 2D-space) as $A(R) = P(R) + M(R)$ to be normalized with respect to the average area occupied by the computation and memory tiles. Then, our definition of efficiency follows from the well-known relation of efficiency E and speedup S:

$$\begin{aligned} E(A,T,N) &= \frac{\mathrm{S}(A)}{A} \\ &= \frac{T(1)/T(A)}{A} \\ &= \frac{C(N)}{A \cdot T(A)} \\ &= \frac{C(N)}{AT}. \end{aligned}$$

The third line follows from the fact that the computational work $C(N)$ equals the number of time steps needed to execute an algorithm with problem size N on a single tile, that is $C(N) = T(1)$. In other words, for a given algorithm and problem size N, efficiency is inversely proportional to the space-time product AT, as stated in Definition 2.

For all practical purposes, we may relate the problem size N and network size R via a real-valued σ such that $N = \sigma R$. Substituting σR for N in Equation 1, we obtain a sufficient condition for computational efficiency as follows.

Definition 3 ***(Computation-Efficiency Condition)***
We call an algorithm with problem size N ***computationally efficient*** *when executed on a network of size R, if and only if*

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

where $N = \sigma R$.

Informally, the computation-efficiency condition demands that we obtain $100\,\%$ computational efficiency asymptotically for an infinitely large network size, and a problem size that is infinitely

larger than the network size.[3] Note that the computation-efficiency condition presents a departure from conventional parallel algorithm design where scalability has been the foremost goal. In particular in the past [13, 31], the design of parallel algorithms has been associated with the assumption that we should be able to utilize larger numbers of tiles as we increase the problem size, and ideally use $P = \Theta(N)$ tiles efficiently. The computation-efficiency condition abandons this criterion in favor of the perspective that real machines are of finite size. Loosely speaking, we prefer to use a small machine very efficiently rather than a large machine less efficiently.

Based on our definitions of the computation-efficiency condition and the decoupling-efficiency condition, we now define a stream algorithm.

Definition 4 ***(Stream Algorithm)***
*A **stream algorithm** is a computation-efficient and decoupling-efficient decoupled systolic algorithm.*

This definition of a stream algorithm suggests that we can express the efficiency of stream algorithms as a product of two terms E_σ capturing computation efficiency and E_R capturing decoupling efficiency, with the former depending on σ and the latter on R only:

$$E(\sigma, R) = E_\sigma(\sigma) \cdot E_R(R). \tag{2}$$

We require that $0 < E_\sigma, E_R \leq 1$, so that $0 < E \leq 1$. The E_σ-term is a rational function representing the *average utilization of the computation tiles* when executing a decoupled systolic algorithm. For the stream algorithms we have developed so far, we find that E_σ assumes the form of a rational function in σ, see Table 1 in Section 8.1:

$$E_\sigma(\sigma) \quad = \quad \frac{\sum_{n=0}^{\hat{n}} u_n \sigma^n}{\sum_{m=0}^{\hat{m}} v_m \sigma^m}, \qquad \text{where} \quad 0 \leq \hat{n} = \hat{m}. \tag{3}$$

The E_R-term represents the *area efficiency* of a decoupled systolic algorithm, that is the ratio of computation tiles $P(R)$ and total number of tiles $P(R) + M(R)$. For the stream algorithms listed in Table 1, E_R is a rational function in R of the form:

$$E_R(R) \quad = \quad \frac{R}{R+a}, \qquad \text{where} \quad a \in \{2, 3, 4\}. \tag{4}$$

Both terms exhibit a behavior that is characteristic for stream algorithms:

$$\lim_{\sigma \to \infty} E_\sigma(\sigma) = 1 \qquad \text{and} \qquad \lim_{R \to \infty} E_R(R) = 1,$$

so that the product of the two terms, the computational efficiency, approaches $100\,\%$ for large values of σ and R.

Figure 5 illustrates the general behavior of efficiency as a function of network size R for the case where $\sigma = R$ or $N = R^2$. The top curve corresponds to the E_σ-term, which approaches its limit 1 very rapidly. The curve below shows the corresponding E_R-term, and the bottom curve is the product of the two terms. Since the E_σ-term approaches 1 so rapidly, the efficiency curve *clamps* onto the curve of the dominating E_R-term.

3. We could write the latter condition as $R = o(N)$ instead of using the limit $\sigma \to \infty$. The little-o condition is quite strong, and it remains an open question whether the weaker demand $R = \Omega(N)$ might be useful.

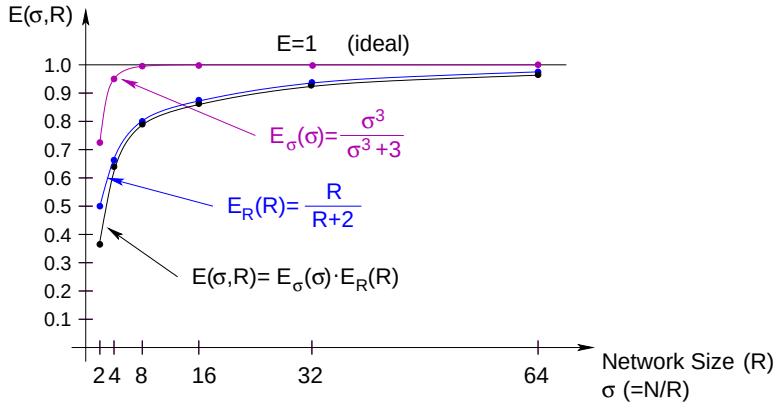


Figure 5: Illustration of efficiency $E(\sigma, R)$ as a function of network size R and $\sigma = N/R$ for the special case $\sigma = R$, that is $N = R^2$. Function $E_\sigma(\sigma)$ represents average utilization of the computation tiles and $E_R(R)$ area efficiency.

In the following sections we discuss the transformation of a matrix multiplication, a triangular solver, and a convolution into stream algorithms. All three designs qualify as stream algorithms because they fulfill the decoupling-efficiency and computation-efficiency conditions. Readers less interested in the technical details of these algorithms and their analysis may skip Sections 4–6.

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:

$$c_{ij} \quad = \quad \sum_{k=1}^{N} a_{ik} \cdot b_{kj}, \tag{5}$$

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

PARTITIONING

Since we are interested in problems of size $N > R$, we start by partitioning the problem into smaller, independent subproblems. Each of the dominating subproblems must execute systolic on the DSA with maximum computational efficiency. For the matrix multiplication we use a block-recursive partitioning. We recurse along the rows of A and the columns of B:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} \quad = \quad \begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \end{pmatrix}. \tag{6}$$

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

Our next goal is to move all memory accesses off the critical path by decoupling the computation, such that memory accesses occur on the memory tiles and computational operations on the computation tiles. We begin by observing that each product element c_{ij} can be computed independently of all others by means of Equation 5. In addition, Equation 6 allows us to stream entire rows of A and entire columns of B through the computation tiles. 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 computation tiles. We implement the resulting subproblems as systolic matrix multiplications, illustrated in Figure 6 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.

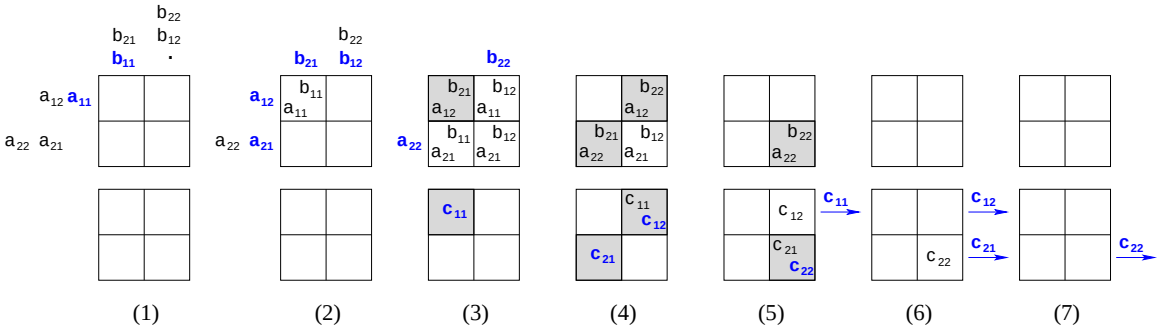


Figure 6: Seven time steps of a systolic matrix multiplication $C = A \cdot B$ for 2×2 matrices. Each box represents a computation tile. Values entering, leaving, or being generated in the array are shown in bold face. Shaded boxes mark the completion of an inner product. We split the data flow of the operands and products into the top and bottom rows.

For $N > R$, the computation tile 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 computation tiles with the proper data streams, we use R memory tiles to store the rows of A and R additional memory tiles to store the columns of B. Thus, for the matrix multiplication, we use $P = R^2$ computation tiles and $M = 2R$ memory tiles. Figure 7 illustrates the data flow of a decoupled systolic matrix multiplication for $N = 4$ and $R = 2$. Note how the memory tiles on the periphery determine the schedule of the computations by streaming four combinations of rows of A and columns of B into the computation tiles. 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 $\{B(:,1),\ B(:,2)\}$, 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 computation tiles or off the chip altogether, but shall be stored in memory tiles, we may have to invest another R memory tiles 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.

EFFICIENCY ANALYSIS

We now analyze the efficiency of our matrix multiplication, and show that it qualifies as a stream algorithm. The number of multiply-and-add operations in the multiplication of two $N \times N$ matrices

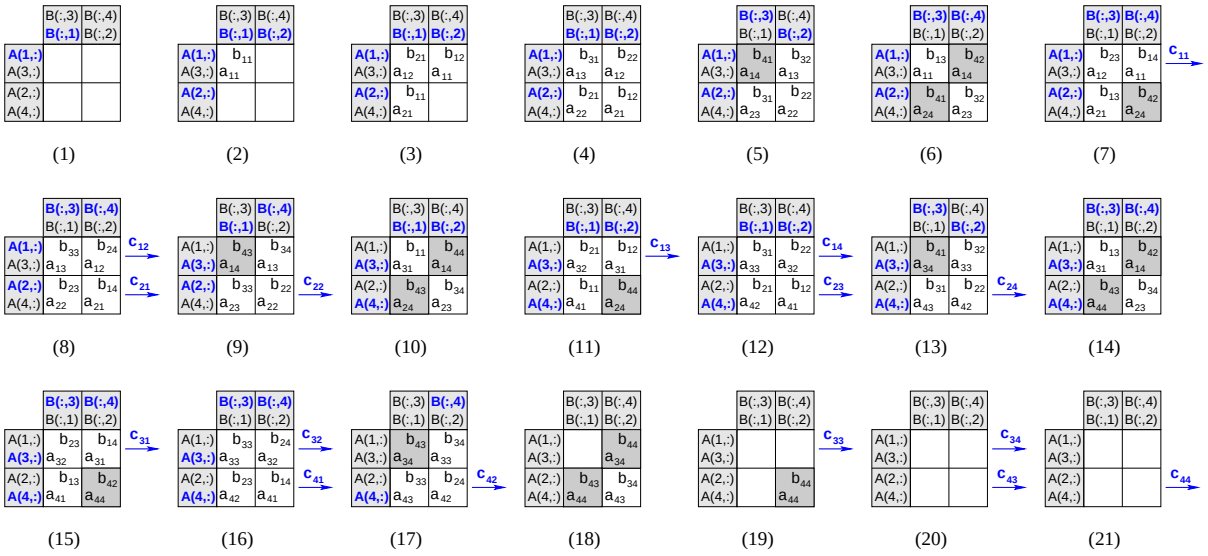


Figure 7: Data flow of a computation-efficient matrix multiplication $C = A \cdot B$ for 4×4 matrices on 2×2 computation tiles. Shaded boxes on the periphery mark memory tiles, and indicate the completion of an inner-product otherwise.

is $C(N) = N^3$. On a network of size R with $P = R^2$ computation tiles and $M = 2R$ memory tiles, 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 tile utilization, and the startup and drain phases combined take $3R$ time steps (cf. Figure 7), 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 computational efficiency of our matrix multiplication is therefore

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

Using $\sigma = N/R$ instead of parameter N, we obtain

$$E_{mm}(\sigma, R) \quad = \quad \frac{\sigma^3}{\sigma^3 + 3} \cdot \frac{R}{R + 2} \tag{7}$$

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. This is the asymptotic behavior illustrated in Figure 5. 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 computational 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 $(N/\sigma)^2$ computation tiles.

To discuss a suitable network size R for a DSA, let us consider some absolute numbers. For example, if $N = R$, that is $\sigma = 1$, we have a systolic matrix multiplication with

$$E_{mm}(\sigma = 1, R) = \frac{1}{4} \cdot \frac{R}{R+2}. \tag{8}$$

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 computationally 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. For a single DSA chip with network size $R = 8$, the computational efficiency of a matrix multiplication of problem sizes $N \geq 64$ is $E(\sigma \geq 8, R = 8) > 79\,\%$.

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 9. Matrices A_{11} and A_{22} are lower triangular.

$$\begin{pmatrix} A_{11} & 0 \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{pmatrix} = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \tag{9}$$

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

$$\begin{aligned} A_{11}X_{11} &= B_{11} && (10)\\ A_{11}X_{12} &= B_{12} && (11)\\ B'_{21} &= B_{21} - A_{21}X_{11} && (12)\\ B'_{22} &= B_{22} - A_{21}X_{12} && (13)\\ A_{22}X_{21} &= B'_{21} && (14)\\ A_{22}X_{22} &= B'_{22} && (15) \end{aligned}$$

First, we compute the solution of the lower-triangular systems in Equations 10 and 11, yielding X_{11} and X_{12}. We use these solutions subsequently to update matrices B_{21} and B_{22} in Equations 12 and 13, producing B'_{21} and B'_{22}. We could compute the matrix subtraction in Equations 12 and 13 on the computation tiles of the array. However, we can save the associated data movement by executing the subtraction on the memory tiles. 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 14 and 15. Solving these systems yields X_{21} and X_{22}. Thus, Equations 10–15 define a recursive algorithm for solving the lower-triangular system of Equation 9. 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, and the computations of columns of X may be performed independently.

Figure 8 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 computation array, while columns of X stream from the bottom of the array. Tile p_{ij} in row i and column j of the computation 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 8 for an upper-triangular solver by reversing the order in which the rows of A are stored on the memory tiles, and by reversing the order in which the elements of the columns of B are fed into the computation tiles.

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 8. Tile 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, tile 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, tile p_{31} receives a_{33}

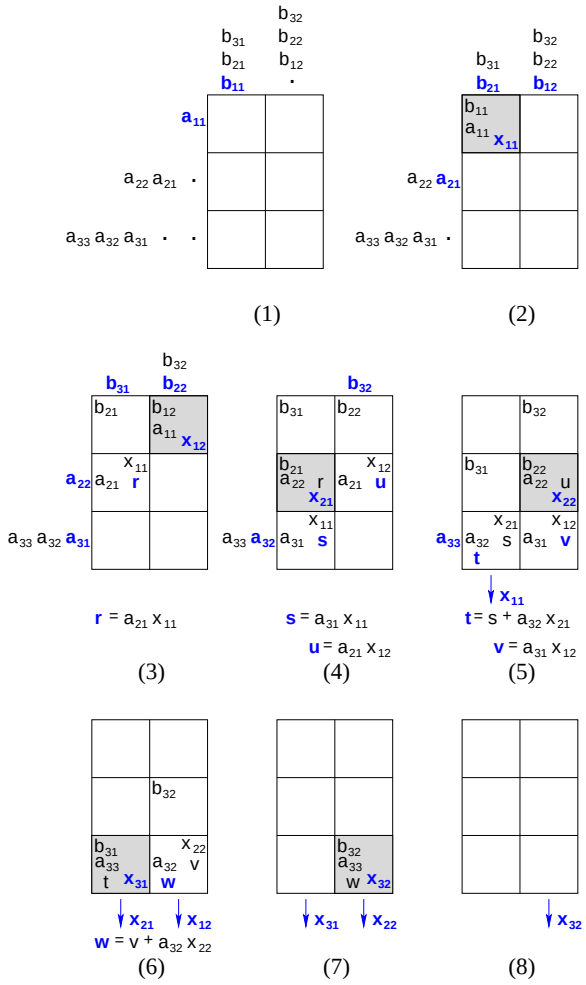


Figure 8: Systolic lower-triangular solver for $N = 3$ and two right-hand sides.

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 computation tiles, we need $3R$ memory tiles on the periphery of the computation array to buffer matrices A, B, and X. Figure 9 shows the computation of X_{11} and X_{21} by means of Equations 10, 12, and 14. As implied by this figure, we use R memory tiles to store the rows of A, and R memory tiles for the columns of B and X, respectively. Thus, for a decoupled systolic lower-triangular solver, we require $P = R^2$ computation tiles and $M = 3R$ memory tiles, 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 10–15. However, we can minimize the loss of cycles by grouping independent computations of the

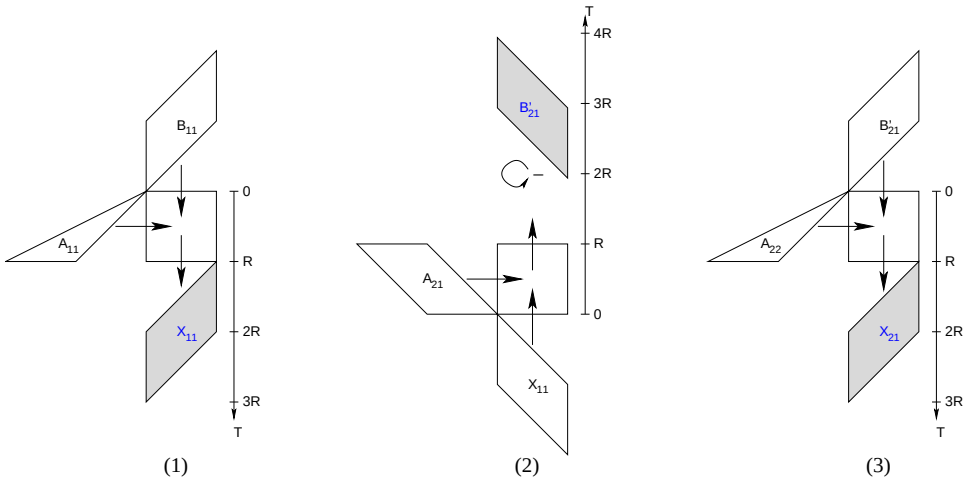


Figure 9: Phases of a decoupled systolic lower-triangular solver on an $R \times R$ array of computation tiles. In phase 1 we solve Equation 10 for X_{11}. In phase 2 we update B_{21} according to Equation 12. While the matrix multiplication is executed on the computation tiles, the matrix subtraction is performed on the memory tiles as they receive each individual result of $A_{21}X_{11}$. Finally, in phase 3 we solve Equation 14 for X_{21}. The shapes of the matrix areas indicate how the rows and columns enter the computation array in a staggered fashion.

same type, and pipelining those before switching to another group. For example, we can group and pipeline the computations of Equations 10 and 11, then Equations 12 and 13, and finally Equations 14 and 15. 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.

EFFICIENCY ANALYSIS

We begin by determining the computation efficiency of our lower-triangular solver according to Equation 1. The number of 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 10–15 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 16, where matrices A, X, and B consist of $\sigma \times \sigma$ blocks, each block is of dimension $R \times R$, and $\sigma = 4$.

$$\begin{pmatrix} A_{11} & 0 & 0 & 0 \\ A_{21} & A_{22} & 0 & 0 \\ A_{31} & A_{32} & A_{33} & 0 \\ A_{41} & A_{42} & A_{43} & A_{44} \end{pmatrix} \begin{pmatrix} X_{11} & X_{12} & X_{13} & X_{14} \\ X_{21} & X_{22} & X_{23} & X_{24} \\ X_{31} & X_{32} & X_{33} & X_{34} \\ X_{41} & X_{42} & X_{43} & X_{44} \end{pmatrix} = \begin{pmatrix} B_{11} & B_{12} & B_{13} & B_{14} \\ B_{21} & B_{22} & B_{23} & B_{24} \\ B_{31} & B_{32} & B_{33} & B_{34} \\ B_{41} & B_{42} & B_{43} & B_{44} \end{pmatrix} \quad (16)$$

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 over-

lap. In the 2×2 partitioning of Figure 9, 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 10–15. Now, consider column block i of the 4×4 example in Equation 16 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 $(N/\sigma)^2$ computation tiles.

According to Equation 1, the computational efficiency of our lower-triangular solver is then

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

Analogous to Equation 7 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 computational efficiency approaches the optimal value of $100\,\%$. Since the solver requires memory tiles along three sides of the array of computation tiles, 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 $R > 27$, our golden network size.

6. Convolution

The convolution of sequence $[a]$ of length $N_{samples}$ with sequence $[w]$ of length N_{taps} produces an output sequence $[b]$ of length $N_{samples} + N_{taps} - 1$. Without loss of generality, we assume that $N_{samples} \geq N_{taps}$. Element k of $[b]$ is given by

$$b_k = \sum_{i+j=k+1} a_i \cdot w_j \tag{17}$$

where

$$\begin{aligned} 1 \leq \quad & k \quad \leq N_{samples} + N_{taps} - 1 \\ 1 \leq \quad & i \quad \leq N_{samples} \\ 1 \leq \quad & j \quad \leq N_{taps}. \end{aligned}$$

PARTITIONING

We partition the convolution into N_{taps}/R subproblems by partitioning the sum in Equation 17 as follows:

$$b_k = \sum_{l=1}^{N_{taps}/R} \sum_{i+j=k+1} a_i \cdot w_j \tag{18}$$

where

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

This partitioning expresses the convolution of $[a]$ and $[w]$ as the sum of convolutions of $[a]$ with N_{taps}/R weight sequences $[w]_j$. Intuitively, we partition weight sequence $[w]$ 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 10 to implement a convolution with $N_{taps} = R$. This design is independent of the length $N_{samples}$ of sequence $[a]$. For the example in Figure 10 we have chosen $N_{taps} = R = 4$ and $N_{samples} = 5$. Both sequence $[a]$ and weight sequence $[w]$ enter the array from the left, and output sequence $[b]$ leaves the array on the right. Computation tile p_i is responsible for storing element w_i of the weight sequence. Thus, the stream of elements w_i folds over on the way from left to right through the array. In contrast, sequence $[a]$ streams from left to right without folding over. During each time step, the computation tiles 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.

We illustrate the data movement in Figure 10 by discussing the computation of $b_4 = a_4w_1 + a_3w_2 + a_2w_3 + a_1w_4$. We begin with time step 5 in Figure 10 when element a_4 enters tile p_1 on

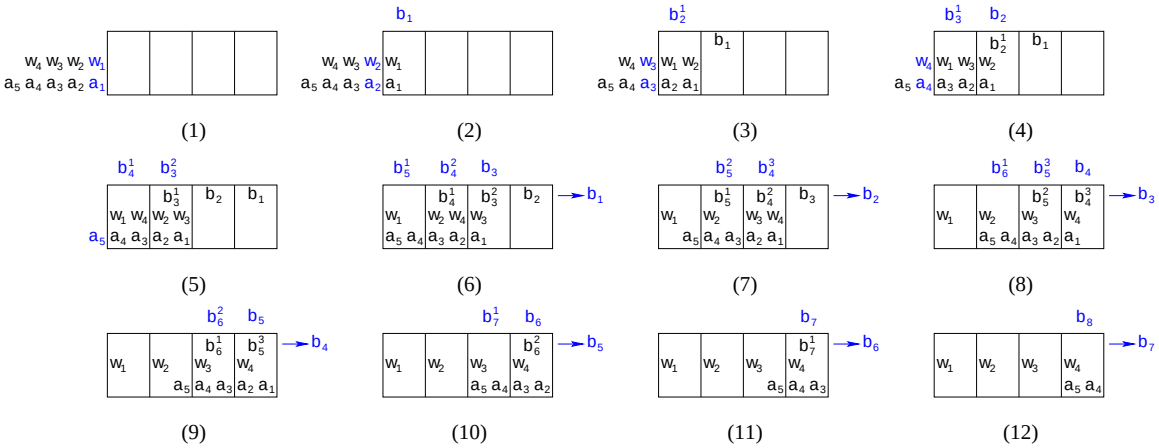


Figure 10: Systolic convolution of a sequence of input values a_i of length $N_{samples} = 5$ with $N_{taps} = 4$ weights w_j. Both the weights and input sequence are fed into the linear array of $R = 4$ computation tiles. Intermediate results are shown above the corresponding tiles. Value b_k^i represents an intermediate value of b_k after the first i products have been computed according to Equation 17.

the left. Element w_1 is already resident. Tile p_1 computes the intermediate value $b_4^1 = a_4 \cdot w_1$, and sends it to tile p_2. At time step 6, p_2 receives a_3 and b_4^1 from tile p_1 on the left. With weight w_2 already resident, tile 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 tile p_3. It computes and sends intermediate value $b_4^3 = b_4^2 + a_2 \cdot w_3$ toward tile 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 computation array.

We use the partitioning of Equation 18 to reduce a convolution with a weight sequence of length N_{taps} into N_{taps}/R systolic convolutions that match network size R of a linear array of computation tiles. In addition, we employ one memory tile on the left of the array to buffer sequences $[a]$ and $[w]$, and another memory tile on the right of the array to store intermediate values of the computation as well as to compute the sum of the subproblems. Figure 11 illustrates the computation of a convolution on a linear processor array. Our decoupled systolic convolution requires $P = R$ computation tiles and $M = 2$ memory tiles. We observe that $M = o(P)$ and, therefore, our convolution is decoupling efficient. Note that the organization in Figure 11 requires three networks between the computation tiles. Sacrificing an asymptotically insignificant amount of efficiency, we can preload the weights in order to reduce the number of needed networks to the two supported by our decoupled systolic architecture.

EFFICIENCY ANALYSIS

The number of multiply-and-add operations for the convolution of a sequence $[a]$ of length $N_{samples}$ with a weight sequence $[w]$ of length N_{taps} is $C(N_{samples}, N_{taps}) = N_{samples} N_{taps}$. On a linear network of size R with $P(R) = R$ computation tiles and $M = 2$ memory tiles, we partition the computation into $\sigma = N_{taps}/R$ subproblems, each of which is a convolution of sequence $[a]$ of

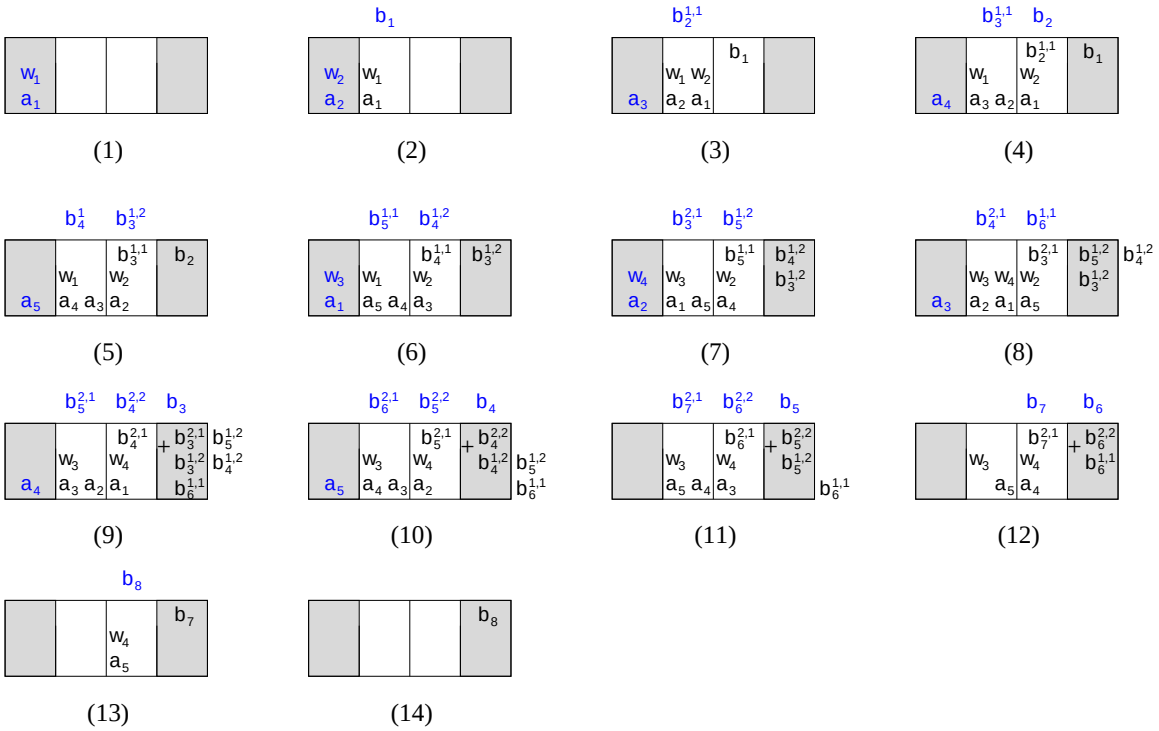


Figure 11: Stream convolution of an input sequence of length $N_{samples} = 5$ with $N_{taps} = 4$ weights on a linear array of $R = N_{taps}/2 = 2$ computation tiles and $M = 2$ memory tiles. Value $b_k^{l,i}$ represents the computation of b_k when the outer summation of Equation 18 has been executed 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.

length $N_{samples}$ with weight sequence $[w]$ of length R. These subproblems overlap perfectly, as is obvious from Figure 11.

We account for the time steps of the convolution as follows. There are $\sigma = N_{taps}/R$ systolic convolutions on a linear array of R computation tiles that pipeline perfectly. Each of the systolic convolutions requires $N_{samples} + R$ time steps to stream a sequence of length $N_{samples}$ 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:

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

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

$$E_{conv}(\sigma, R) \quad = \quad \frac{\sigma^2}{\sigma^2 + N_{taps}/N_{samples}} \cdot \frac{R}{R+2}.$$

Given our assumption that $N_{samples} \geq N_{taps}$, we have $N_{taps}/N_{samples} \leq 1$, 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 $N_{taps} = R$ or, equivalently, $\sigma = 1$, the stream convolution reduces to a systolic convolution with a computational efficiency of

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

We note that for $N_{taps} = N_{samples}$ 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.

7. Related Work

In the following, we compare our work on stream algorithms and architecture with some of the prominent contributions in the arena of systolic computing. The primary novelty of our paper is a rigorous yet simple definition of stream algorithms as an asymptotically optimal mapping of a decoupled systolic algorithm into a decoupled systolic architecture (DSA). As a programmable tiled microarchitecture the DSA revives numerous concepts invented earlier for systolic architectures [22, 32, 33, 34, 35]. In fact, the DSA by itself is not a particularly novel architecture. Rather, it embodies a minimal selection of architectural features motivated by the analysis of stream algorithms that allows us to execute stream algorithms in a flexible, area efficient, computationally efficient, and energy efficient manner. In addition, the DSA can be viewed as an abstract machine for emulating stream computations on existing tiled microarchitectures.

The roots of the DSA lie in the earliest systolic arrays which were inspired by the design philosophy expounded by Kung and Leiserson [10]. These arrays lacked flexibility because they were special-purpose circuits for a particular application with a particular problem size. Such specialized designs are still in use to date for ultra high performance digital signal processing, for example. Programmable systolic systems such as the Wavefront Array Processor [32], PSC [34], Warp [22], iWarp [35], and Saxpy's Matrix-1 [33] offered flexibility including independence from problem size, but did not realize the potential of stream algorithms. For example, although H.T. Kung advocated *systolic communication* [36] as a means for achieving high efficiency by avoiding local memory accesses, reports about systolic algorithms [22, 35, 36, 12, 37] show that these implementations do actually use local memory, whereas our stream algorithmic versions do not. *Simulation* [13] or *emulation* [38] of multiple processing elements on one larger, more powerful processor have become the parallel programming methodology of choice, because they allow for problem-size independence and network embedding, albeit at the cost of potentially unbounded local memory requirements. As a consequence of local memory accesses, simulated algorithms are slower than stream algorithms (on load-store architectures typically by a factor of three), consume significantly more energy due to memory accesses, and occupy a larger silicon area than the DSA.

The primary novelty of stream algorithms lies in our efficiency definitions which serve as a guiding methodology allowing us to qualify stream algorithms as those among the larger class of decoupled systolic algorithms that achieve $100\,\%$ efficiency asymptotically. In particular, our stream algorithms achieve higher efficiencies than most purely systolic designs. As a first case in point, consider the computational efficiency of a systolic matrix multiplication on a special-purpose systolic array that matches the problem size. Due to startup and drain phases of the staggered dataflow (see

Figure 6), the computational efficiency of a systolic matrix multiplication is just $25\,\%$ according to Equation 8, which also accounts for streaming the result matrix out of the array. Similarly, most systolic algorithms proposed in the past [10, 11] suffer from bubbles in the data streams in favor of a simple processor design. Our definition of stream algorithms prescribes a large problem size compared to the network size of the decoupled systolic architecture. It is due to this requirement that our stream matrix multiplication achieves $100\,\%$ computational efficiency. Of course, a special-purpose array can also achieve $100\,\%$ computational efficiency in a scenario where multiple independent matrix multiplications can be overlapped or interleaved. However, more complex applications with phases of different systolic algorithms may not offer the luxury of identical, independent subproblems. For example, the matrix multiplication based on the systolic communication style proposed by H.T. Kung [36, 22, 37, 35] stores one of the matrix operands in local memory and streams the other matrix through the processing elements. This programming style falls short of exploiting the full potential of systolic communication.

As a second case in point, consider the computational efficiency of a simulation [13, 38] which uses one larger processing element to simulate multiple systolic processors and uses local memory for the associated computational state. In the best case, all values fit into local memory but remain subject to loads and stores to and from local memory. For example, a simple single-issue processing element requires up to three loads and one store instruction for a multiply-and-add operation depending on the quality of register allocation. As a consequence, the simulated version is up to four times slower than a stream algorithm that does not use local memory but decouples memory accesses from computation. Of course, we could use a more complex processing element, such as a VLIW architecture, to mask local memory accesses. However, this solution does not provide the area and energy efficiency of the DSA, which achieves high performance with a simple processor core and without local memories.

We can emulate the DSA on programmable tiled microarchitectures with various degrees of efficiency. As a consequence, the design philosophy developed for stream algorithms carries over to tiled architectures as well as programmable systolic array processors. For example, we have emulated the DSA on Raw by implementing stream algorithms in Raw assembly [39]. Since the power supply for Raw's on-chip memories is software controlled, stream algorithms are energy efficient on Raw when powering down the on-chip data memories. However, Raw is not as area efficient as the canonical DSA, which also offers the potential of being operated at higher clock frequencies because the absence of local memory leads to smaller tiles and shorter wires.

Our design methodology for stream algorithms [14] includes an application specific partitioning step. Partitioning by itself is not new. For example, in the context of systolic arrays, Navarro et al. [11] discuss and classify partitioning schemes as *spatial mappings* and *temporal mappings.* Their notion of spatial mapping coincides with that of simulating multiple systolic processing elements on a more powerful one. The temporal mappings proposed in [11] are closer to stream algorithms in that they partition systolic algorithms into heterogeneous phases, although their methodology differs from ours in that it is centered around the idea of transforming dense matrix operations into band matrix operations. Moreover, the temporal mappings introduced in [11] are neither computationally efficient nor decoupling efficient in the sense defined in Section 3. Lack of attention to partition applications with decoupling efficiency in mind prevents these systolic algorithms from enjoying the energy-efficiency advantage of stream algorithms. It is noteworthy that the blocked matrix computations at the heart of the Saxpy Matrix-1 programming philosophy are a type of simulation or spatial mapping that are not decoupling efficient either.

8. Results

In this section, we summarize our results related to the design of stream algorithms, and present two practical results about stream algorithms and architecture. First, we present experimental results from emulating the DSA on an existing tiled architecture, demonstrating the benefits of stream algorithms as a design discipline. Second, we argue that a reasonable network size for a single-chip DSA is $R = 8$.

8.1 Summary of Stream Algorithms

Table 1 summarizes the stream algorithms that we have developed thus far. For each of the stream algorithms, the table lists the number of computation tiles P and memory tiles M as a function of network size R, and the execution time T and computational efficiency E as functions of R and $\sigma = N/R$ for problem size N. Note that the efficiency $E(\sigma, R)$ approaches $100\,\%$ for $\sigma \to \infty$ and $R \to \infty$. Thus, according to Definition 4, all of the algorithms qualify as stream algorithms.

Application	$P(R)$	$M(R)$	$T(\sigma, R)$	$E(\sigma, R)$
$A \cdot B$	R^2	$2R$	$R(\sigma^3 + 3)$	$\frac{\sigma^3}{\sigma^3+3} \cdot \frac{R}{R+2}$
$A \cdot B^T$	R^2	$3R$	$R(\sigma^3 + 3)$	$\frac{\sigma^3}{\sigma^3+3} \cdot \frac{R}{R+3}$
Triangular solver	R^2	$3R$	$\frac{R}{2}(\sigma^3 + \sigma^2 + 6\sigma - 2)$	$\frac{\sigma^3}{\sigma^3+\sigma^2+6\sigma-2} \cdot \frac{R}{R+3}$
LU factorization	R^2	$3R$	$\frac{R}{6}(2\sigma^3 + 3\sigma^2 + 31\sigma - 12)$	$\frac{\sigma^3}{\sigma^3+\frac{3}{2}\sigma^2+\frac{31}{2}\sigma-6} \cdot \frac{R}{R+3}$
Cholesky fact.	R^2	$3R$	$\frac{R}{6}(\sigma^3 + 3\sigma^2 + 32\sigma - 12)$	$\frac{\sigma^3}{\sigma^3+3\sigma^2+32\sigma-12} \cdot \frac{R}{R+3}$
QR factorization	R^2	$3R$	$\frac{R}{6}(10\sigma^3 + 21\sigma^2 + 125\sigma - 18)$	$\frac{\sigma^3}{\sigma^3+\frac{21}{10}\sigma^2+\frac{25}{2}\sigma-\frac{9}{5}} \cdot \frac{R}{R+3}$
SVD	R^2	$4R$	$\sigma_M^2(5\sigma_N R + 6R + 3)/2 \cdot O(\lg M)$	$\frac{\sigma_N}{\sigma_N+6/5} \cdot \frac{R}{R+4}$
Convolution	R	2	$\sigma N_{samples} + R$	$\frac{\sigma^2}{\sigma^2+N_{taps}/N_{samples}} \cdot \frac{R}{R+2}$
DFT	R	2	$R(\sigma^2 + 2)$	$\frac{\sigma^2}{\sigma^2+2} \cdot \frac{R}{R+2}$
Vandermonde	R	2	$R(\sigma^2 + \sigma + 3)$	$\frac{\sigma^2}{\sigma^2+\sigma+3} \cdot \frac{R}{R+2}$

Table 1: Summary of stream algorithms. We show the number of computation tiles P, number of memory tiles M, and we compare the execution time T and computational efficiency E for problem size N and $\sigma = N/R$ (see [14, 15] for details).

8.2 Emulation Results

We have demonstrated the performance benefits of stream algorithms using a cycle-accurate simulator of the Raw architecture [1] with network size $R = 4$, extended with peripheral memory tiles and without using local data memories. Table 2 compares the computational efficiency of Raw with the predicted efficiency of the decoupled systolic architecture according to Table 1. The computational efficiencies for Raw fall slightly short of those predicted for the DSA when using smaller problem sizes because of a number of architectural reasons discussed in [39]. For large problem

sizes, that is $\sigma \gg 1$, Raw approaches the efficiency of the DSA. Due to the small network size $R = 4$, computational efficiency is dominated by factor E_R in Equation 2.

Application ($R = 4$)	Problem Size N				
	128	256	512	1,024	2,048
$A \cdot B$	0.59 / 0.67	0.62 / 0.67	0.63 / 0.67	0.64 / 0.67	0.64 / 0.67
Tri. Solver	0.34 / 0.55	0.42 / 0.56	0.47 / 0.57	0.50 / 0.57	0.52 / 0.57
LU fact.	0.28 / 0.54	0.37 / 0.56	0.44 / 0.56	0.48 / 0.57	0.51 / 0.57
QR fact.	0.37 / 0.53	0.44 / 0.55	0.48 / 0.56	0.50 / 0.57	0.52 / 0.57
Convolution	0.47 / 0.66	0.55 / 0.67	0.59 / 0.67	0.62 / 0.67	0.63 / 0.67

Table 2: Computational efficiency E_{raw}/E_{dsa} of stream algorithms. We compare the efficiency E_{raw} of the Raw microprocessor with network size $R = 4$ with the predicted efficiency E_{dsa} of our decoupled systolic architecture. In case of the convolution, we report results for $N_{samples} = N$ and $N_{taps} = 32$.

8.3 Chip Size

In order to implement a single-chip DSA, we must choose a particular network size R. In Section 2.3, we have seen that $R = 8$ allows us to design a 32 bit architecture without being limited by the pin constraints of today's packaging technology. In this section, we provide two additional arguments in favor of $R = 8$.

Our first argument is based on the analysis of stream algorithms. Recall from Equation 2 that we may express the efficiency of a stream algorithm as a product of two terms E_σ and E_R. According to Equation 4, the E_R-term depends on network size R only. In Equation 4 we introduced the algorithm specific parameter a, and interpret it as the number of peripheral sides of the computation array that data streams cross on the way in and out of memory tiles. For the stream algorithms listed in Table 1, we find that, on average, the number of sides is $a = 3$. The maximum value $a = 4$ corresponds to the four sides of a two-dimensional mesh of computation tiles.

We declare the ***golden network size*** R^* to be obtained when the E_R-term assumes the $90\,\%$ mark for $a = 3$:

$$E_R(R) = 0.9 \qquad \Rightarrow \qquad R^* = 27.$$

In other words, if the E_σ-term approaches value 1, which it does for reasonably large problem sizes, then most stream algorithms will achieve $90\,\%$ computational efficiency on a DSA with network size $R^* = 27$. In contrast, for network size $R = 8$, the maximum computational efficiency for stream algorithms with $a = 3$ is $E = 73\,\%$. In fact, this efficiency level is satisfactory on its own right, allowing us to argue that $R = 8$ constitutes a good network size for a single chip, which can serve as building block for larger systems. A fat tree of 16 chips of network size $R = 8$ would behave like a 32×32 DSA, exceeding the golden network size $R^* = 27$.

While the golden network size resembles a lower-bound argument for network size R, our second argument provides an upper bound. If we are interested in expanding the decoupled systolic

architecture to serve as a general-purpose multiprocessor, we may want to run multiple applications concurrently. In the worst case, these applications will be strictly sequential and data intensive, and should be executed on the peripheral computation tiles next to memory tiles. This raises the question: what is the largest sensible network size R, if we do not utilize the inner computation tiles at all, but do still desire a respectable computational efficiency. A back-of-the-envelope estimate attempts to answer this question. We define the following variables:

R**:** network size of $R \times R$ array of computation tiles,

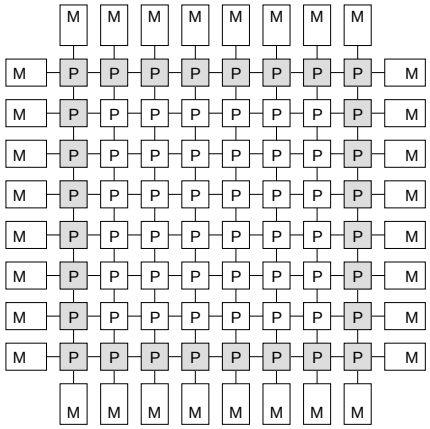
P**:** number of peripheral computation tiles ($P = 1$ for $R = 1$ and $P = 4(R-1)$ for $R > 1$),

I**:** number of idling computation tiles ($I = R^2 - P$),

E**:** maximum efficiency if P peripheral tiles perform work and are fully utilized, while the I inner tiles idle ($E = P/R^2$).

Figure 12 illustrates an $R \times R$ array of computation tiles for $R = 8$. The peripheral computation tiles are shaded gray. There are $P = 4(8-1) = 28$ working computation tiles. The remaining computation tiles shall be idle. Their number is $I = 8^2 - 28 = 36$. Then, the maximum efficiency achievable by any set of applications is $E = 28/8^2 = 44\,\%$ with respect to the number of computation tiles.

The table on the right of Figure 12 shows the maximum efficiency E as a function of network size R. Since E is inversely proportional to R, the efficiency drops quickly as R increases. However, for $R = 8$ we still obtain $44\,\%$ efficiency, that is close to $50\,\%$. In addition, for $R = 8$ we are also able to harvest almost $50\,\%$ of the peak performance of the entire chip. Losing no more than a factor of 2 is certainly acceptable, given that most general scheduling problems are considered near optimal when reaching a factor of 2 from the optimum. Consequently, we consider network size $R = 8$ as a very reasonable upper bound for the size of a single-chip DSA.



R	P	I	$E\,[\%]$
1	1	0	100
2	4	0	100
4	12	4	75
8	28	36	44
16	60	196	23
32	124	900	12
64	252	3844	6

Figure 12: An $R \times R$ decoupled systolic architecture for $R = 8$. Peripheral computation tiles are shaded gray. The table on the right shows the maximum sustainable efficiency ($E = P/R^2$) as a function of R when utilizing $P = 4(R-1)$ peripheral tiles only.

In practice, the network size of a DSA chip will have to fit on the area provided by an existing VLSI process. Our estimates based on the Raw chip [1] indicate that an 8×8 DSA chip can be

built with a $0.09\mu m$ process. Such a process should permit driving the chip at clock frequencies up to 10 GHz, so that the DSA would *sustain* a performance of nearly 640 Gflops on stream algorithms. This level of performance is quite remarkable. We would be able to shrink Japan's Earth simulator, that occupies an entire building with its own power plant, to about 10 chips and IBM's Blue Gene supercomputer to about 100 chips. In addition, a DSA would enable unconventional high-performance applications, including any type of *software radio* such as HDTV to execute in real time.

Finally, let us emphasize that 8×8 is a reasonable size of a computation array with peripheral memory as an architectural building block, that is a *mega tile*, of a larger decoupled systolic architecture. If the future permits us to build chips with an area capacity to accommodate more than one 8×8 array, it should consist of multiple 8×8 arrays, each with its own peripheral memory. It is perfectly reasonable to have a network whose size exceeds $R = 8$ and even the golden network size R^*, because stream algorithms become more efficient as we increase the network size R.

9. Conclusions

We have studied the feasibility of highly efficient computation on tiled microarchitectures. As a result, we propose a design of stream algorithms and architecture that permits $100\,\%$ computational efficiency asymptotically for large networks and appropriate problem size. As a consequence, our decoupled systolic architecture interprets stream algorithms in an energy efficient manner by design.

We have transformed a number of regular applications into stream algorithms. Besides the three examples presented in Sections 4–6, we have summarized the stream algorithms developed thus far in Table 1 of Section 8.1. Our design methodology for stream algorithms has led us to discover new ground. For instance, to our best knowledge, our stream QR [14] and stream SVD [15] appear to be the most efficient parallel organizations of these algorithms in existence to date.

To achieve highest efficiency levels, we have developed application-specific partitioning and decoupling transformations, that are comparable in complexity to those needed for the design of systolic arrays [10]. We believe that it is unreasonable to expect conventional compiler technology to generate competitive code, not even when based on advanced methods for automatic parallelization and targeting systolic arrays [40, 41, 42]. Instead, we feel that template-based code generation à la FFTW [43] is the most promising way to compile stream algorithms.

A pleasant side effect of highly efficient computation is the predictability of the number of execution cycles. Our efficiency analysis of stream algorithms allows us to predict computational performance even in the case where network and problem size are too small to approach the optimal computational efficiency of $100\,\%$. Performance predictability, in turn, should enable a reasonably accurate prediction of power consumption. Thus, if performance is a good indicator for power consumption, we may employ a circuit design for the decoupled systolic architecture that supports varying clock frequencies to trade power consumption for computational performance. Programmable microprocessors, in contrast to special-purpose hardware, allow us to differentiate quality of service based on performance. For instance, typical algorithms for digital signal processing that deliver a higher quality audio or video require a larger number of operations than others. In this situation, quality is related to computational performance and, thus, to power consumption and clock frequency. It is in light of these physical relations that we emphasize our desire to schedule computations with $100\,\%$ computational efficiency on single-chip microarchitectures.

We view the decoupled systolic architecture as an extreme point in the design space of tiled microarchitectures. It demonstrates that programmable, regular hardware structures can be utilized for high-performance computing in an energy efficient manner. From this perspective, the decoupled systolic architecture serves as a measuring rod for general-purpose microarchitectures that matches the physical and technological constraints of future single-chip designs.

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 express our appreciation to Janice McMahon and Bob Bond for their support.

References

[1] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffmann, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs," *IEEE Micro*, vol. 22, pp. 25–36, March/April 2002.

[2] R. Nagarajan, K. Sankaralingam, D. C. Burger, and S. W. Keckler, "A Design Space Evaluation of Grid Processor Architectures," in *34th Annual International Symposium on Microarchitecture*, pp. 40–51, Dec. 2001.

[3] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic, "The Vector-Thread Architecture," in *31st International Symposium on Computer Architecture*, (München, Germany), pp. 52–63, June 2004.

[4] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "WaveScalar," in *36th International Symposium on Microarchitecture*, (San Diego, CA), pp. 291–302, IEEE Computer Society, Dec. 2003.

[5] J. Oliver, R. Rao, P. Sultana, J. Crandall, E. Czernikowski, L. W. Jones IV, D. Franklin, V. Akella, and F. T. Chong, "Synchroscalar: A Multiple Clock Domain, Power-Aware, Tile-Based Embedded Processor," in *31st International Symposium on Computer Architecture*, (München, Germany), pp. 150–161, June 2004.

[6] T. D. Burd, *Energy-Efficient Processor System Design*. PhD thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 2001.

[7] R. Gonzalez and M. Horowitz, "Energy Dissipation in General Purpose Microprocessors," *IEEE Journal of Solid-State Circuits*, vol. 31, pp. 1277–1284, Sept. 1996.

[8] J. S. Kim, M. B. Taylor, J. Miller, and D. Wentzlaff, "Energy Characterization of a Tiled Architecture Processor with On-Chip Networks," in *ACM Symposium on Low Power Electronics and Design*, (Seoul, Korea), pp. 424–427, Aug. 2003.

[9] J. E. Smith, "Decoupled Access/Execute Computer Architectures," *ACM Transactions on Computer Systems*, vol. 2, pp. 289–308, Nov. 1984.

[10] H. T. Kung and C. E. Leiserson, "Algorithms for VLSI Processor Arrays," in *Introduction to VLSI Systems* (C. A. Mead and L. A. Conway, eds.), ch. 8.3, pp. 271–292, Addison-Wesley, 1980.

[11] J. J. Navarro, J. M. Llaberia, and M. Valero, "Partitioning: An Essential Step in Mapping Algorithms Into Systolic Array Processors," *IEEE Computer*, vol. 20, pp. 77–89, July 1987.

[12] H. T. Kung, "Warp Experience: We Can Map Computations Onto a Parallel Computer Efficiently," in *2nd International Conference on Supercomputing*, pp. 668–675, ACM Press, 1988.

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

[14] H. Hoffmann, V. Strumpen, and A. Agarwal, "Stream Algorithms and Architecture," Technical Memo MIT-LCS-TM-636, Laboratory for Computer Science, Massachusetts Institute of Technology, Mar. 2003.

[15] V. Strumpen, H. Hoffmann, and A. Agarwal, "A Stream Algorithm for the SVD," Technical Memo MIT-LCS-TM-641, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Oct. 2003.

[16] R. Ho, K. W. Mai, and M. A. Horowitz, "The Future of Wires," *Proceedings of the IEEE*, vol. 89, pp. 490–504, Apr. 2001.

[17] G. D. Micheli, *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, 1994.

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

[19] B. R. Rau and J. A. Fisher, "Instruction-Level Parallel Processing: History, Overview and Perspective," *The Journal of Supercomputing*, vol. 7, pp. 9–50, May 1993.

[20] H. T. Kung, "Why Systolic Architectures?," *IEEE Computer*, vol. 15, pp. 37–46, Jan. 1982.

[21] J. A. B. Fortes and B. W. Wah, "Systolic Arrays—From Concept to Implementation (Guest Editors' Introduction)," *IEEE Computer*, vol. 20, pp. 12–17, July 1987.

[22] M. Annaratone, E. Arnould, T. Gross, H. T. Kung, M. S. Lam, O. Menzilcioglu, K. Sarocky, and J. A. Webb, "Warp Architecture and Implementation," in *13th Annual Symposium on Computer Architecture*, pp. 346–356, 1986.

[23] D. S. Henry and C. F. Joerg, "A Tightly-coupled Processor-Network Interface," in *5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 111–122, ACM Press, 1992.

[24] D. W. Hillis, *The Connection Machine.* Cambridge, MA: MIT Press, 1985.

[25] M. Fillo, S. W. Keckler, W. J. Dally, N. P. Carter, A. Chang, Y. Gurevich, and W. S. Lee, "The M-Machine Multicomputer," in *28th International Symposium on Microarchitecture*, (Anne Arbor, MI), pp. 146–156, IEEE Computer Society, Dec. 1995.

[26] C. E. Leiserson, "Fat-Trees: Universal Networks For Hardware-Efficient Supercomputing," *IEEE Transactions on Computers*, vol. 34, pp. 892–901, Oct. 1985.

[27] V. Strumpen and A. Krishnamurthy, "A Collision Model for Randomized Routing in Fat-Tree Networks," Technical Memo MIT-LCS-TM-629, Laboratory for Computer Science, Massachusetts Institute of Technology, July 2002.

[28] S. Palacharla and R. E. Kessler, "Evaluating Stream Buffers as a Secondary Cache Replacement," in *21st International Symposium on Computer Architecture*, pp. 24–33, IEEE Computer Society Press, 1994.

[29] D. F. Zucker, R. B. Lee, and M. J. Flynn, "Hardware and Software Cache Prefetching Techniques for MPEG Benchmarks," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 10, pp. 782–796, Aug. 2000.

[30] C. D. Thompson, "Area-Time Complexity for VLSI," in *11th Annual ACM Symposium on Theory of Computing*, pp. 81–88, 1979.

[31] V. Kumar and A. Gupta, "Analysis of Scalability of Parallel Algorithms and Architectures: a Survey," in *5th International Conference on Supercomputing*, pp. 396–405, ACM Press, 1991.

[32] S.-Y. Kung, R. J. Gal-Ezer, and K. S. Arun, "Wavefront Array Processor: Architecture, Language and Applications," in *Conference of Advanced Research in VLSI*, (M.I.T.), pp. 4–19, Jan. 1982.

[33] D. E. Foulser and R. Schreiber, "The Saxpy Matrix-1: A General-Purpose Systolic Computer," *IEEE Computer*, vol. 20, pp. 35–43, July 1987.

[34] A. L. Fisher, H. T. Kung, L. M. Monier, and Y. Dohi, "Architecture of the PSC—A Programmable Systolic Chip," in *10th International Symposium on Computer Architecture*, pp. 48–53, IEEE Computer Society Press, 1983.

[35] T. Gross and D. R. O'Hallaron, *iWARP: Anatomy of a Parallel Computing System*. Cambridge, MA: MIT Press, 1998.

[36] H. T. Kung, "Systolic Communication," in *International Conference on Systolic Arrays*, (San Diego, CA), pp. 695–703, May 1988.

[37] T. Gross, S. Hinrichs, D. R. O'Hallaron, T. Stricker, and A. Hasegawa, "Communication Styles for Parallel Systems," *IEEE Computer*, vol. 27, pp. 34–44, Dec. 1994.

[38] R. R. Koch, F. T. Leighton, B. M. Maggs, S. B. Rao, A. L. Rosenberg, and E. J. Schwabe, "Work-Preserving Emulations of Fixed-Connection Networks," *Journal of the ACM*, vol. 44, pp. 104–147, Jan. 1997.

[39] H. Hoffmann, "Stream Algorithms and Architecture," Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, June 2003.

[40] R. M. Karp, R. E. Miller, and S. Winograd, "The Organization of Computations for Uniform Recurrence Equations," *Journal of the ACM*, vol. 14, pp. 563–590, July 1967.

[41] S. K. Rao and T. Kailath, "Regular Iterative Algorithms and their Implementation on Processor Arrays," *Proceedings of the IEEE*, vol. 76, pp. 259–269, Mar. 1988.

[42] W. Shang and J. A. B. Fortes, "Time Optimal Linear Schedules for Algorithms with Uniform Dependencies," *IEEE Transactions on Computers*, vol. 40, pp. 723–742, June 1991.

[43] M. Frigo and S. G. Johnson., "FFTW: An adaptive software architecture for the FFT," in *IEEE International Conference on Acoustics Speech and Signal Processing*, vol. 3, pp. 1381–1384, 1998.