Lecture Notes for the Software Track of the PMPH Course

Programming Massively Parallel Hardware (PMPH)

COSMIN E. OANCEA, University of Copenhagen, Denmark

In simple words, the aim of the PMPH course is to teach students how to write programs that run fast on highly-parallel hardware, such as general-purpose graphics processing units (GPGPUs), which are now mainstream. Such architectures are however capricious; unlocking their power requires understanding their design principles and also specialized knowledge of code transformations, for example aimed at optimising locality of reference, the degree of parallelism, etc.

This document consists of the lecture notes for the software track of the PMPH course. The main goal is to teach students how to *think parallel*. In short, we will introduce the map-reduce functional programming model, which builds programs naturally, like puzzles, from a nested composition of implicitly-parallel array operators, which are rooted in the mathematical structure of list homomorphisms.

We will reason about the asymptotic (work and depth) properties of such programs, and discuss the flattening transformation, which converts (all) arbitrarily-nested parallelism to a more-restricted form that can be directly mapped to the hardware.

We then will turn our attention to legacy-sequential code written in programming languages such as C. In this context we study dependence analysis, as a tool for reasoning about loop-based optimizations (e.g., is it safe to execute a given loop in parallel, or to interchange two loops?).

As time permits, we may cover more advanced topics, for example related to various static and dynamic analyses aimed at optimizing locality of reference, communication, or at extracting automatically parallelism from sequential, loop-based code.

Additional Key Words and Phrases: parallelism, compilers, flattening transformation, dependence analysis, dynamic analysis, GPU

Author's address: Cosmin E. Oancea, DIKU, University of Copenhagen, Universitetparken 5, Copenhagen, 2100, Denmark, cosmin.oancea@diku.dk.

2018. 2475-1421/2018/9-ART1 $15.00
https://doi.org/

Contents

Abstract 1
Contents 2
1 Motivation, Hardware Trends and Technological Constraints 4
1.1 Abstractions 5
1.2 Processor Frequency and Number of Transistors 5
1.3 Memory Wall! Which Memory Wall? 7
1.4 Technological Constraints 8
1.4.1 Power 8
1.4.2 Reliability 9
1.4.3 Wire Delays 10
1.4.4 Design Complexity 10
1.4.5 CMOS Meets Quantum Physics 10
2 List Homomorphism (LH) 11
2.1 Math Preliminaries: Monoid and Homomorphism 11
2.2 The Shape of a List-Homomorphic Function/Implementation 11
2.3 Examples of List-Homomorphic Implementations 13
2.4 Map, Reduce and List Homomorphism Theorems 14
2.4.1 First List-Homomorphism Theorem 15
2.4.2 Other List-Homomorphism Lemmas 16
2.5 Almost/Near Homomorphisms [Gorlatch/Cole] 18
2.5.1 Maximum-Segment Sum (MSS) Problem 18
2.5.2 Longest-Satisfying Segment (LSS) Problem 20
2.5.3 Exercise: Longest-Satisfying Segment (LSS) Implementation 21
2.6 Conclusion 22
3 Work-Depth Asymptotic, Nested Parallelism 23
3.1 Reasoning in Terms of Work-Depth Asymptotic 23
3.1.1 Amdahl's Law 23
3.1.2 Work-Depth Asymptotic Behavior of a Parallel Program 26
3.1.3 Demonstrating Work-Depth Computation for Reduction 26
3.1.4 Naive and Native Implementation of Reduction in Futhark 28
3.2 Other Parallel Operators, Examples of Nested-Parallel Applications 31
3.2.1 Types and Semantics of Various Parallel Operators 32
3.2.2 Implementation of Scan and Segmented Scan 34
3.2.3 Implementation of Partition (Filter) 39
3.2.4 Sparse-Matrix Vector Multiplication 40
3.2.5 Prime Number Computation (Sieve) 41
3.2.6 Quicksort 45
4 The Flattening Transformation 47
4.1 Rules For Flattening 48
4.1.1 Data Flattening (Flat Array Representation) 48
4.1.2 Flattening a Scan Directly Nested in a Map 51
4.1.3 Flattening a Map Directly Nested in a Map 52
4.1.4 Flattening a Replicate Directly Nested in a Map 53
4.1.5 Flattening an Iota Directly Nested in a Map 55
4.1.6 Flattening a Reduce Directly Nested in a Map (Segmented Reduce) 56
4.1.7 Flattening an If-Then-Else Directly Nested in a Map 57

4.1.8 Exercise: Flattening an Index Directly Nested in a Map 58
4.2 Flattening a Simple, Contrived Program 59
4.3 Flattening Exercises 62
4.3.1 Exercise: Flattening Sparse-Matrix Vector Multiplication 62
4.3.2 Exercise: Flattening Prime Number Computation (Sieve) 62
4.3.3 Exercise: Flattening Quicksort 63
5 Loop-Based Data-Dependence Analysis and Applications 66
5.1 Data-Dependence Analysis: Direction Vectors 67
5.1.1 Loop Notation and Lexicographic Ordering of Iterations in a Loop Nest 68
5.1.2 Dependency Definition 69
5.1.3 Aggregating Dependencies by Means of Direction Vectors 69
5.2 Determining Loop Parallelism by Analyzing Direction Vectors 73
5.3 Loop Interchange: Legality and Applications 74
5.4 Loop Distribution: Legality and Applications 75
5.5 Eliminating False Dependencies (WAR and WAW) 78
5.5.1 Eliminating WAR Dependencies by Copying 78
5.5.2 Eliminating WAW Dependencies by Privatization 80
5.6 Recognizing Data-Parallel Operators Hidden in Sequential, Imperative Code 81
5.6.1 Recognizing Reduce Operators 81
5.6.2 Recognizing Scan Operators 83
5.6.3 Recognizing Filter Operators 84
5.7 Loop Stripmining, Block and Register Tiling 84
6 Applications of Data-Dependence Analysis on GPU Hardware 88
6.1 Optimizing the Spatial Locality of Read/Write Accesses on GPUs by Transposition 88
6.2 Transposition: Block Tiling Optimizes Spatial Locality 90
6.3 Matrix-Matrix Multiplication: Block Tiling Optimizes Temporal Locality 93
6.4 Exercise: Block and Register Tiling for Matrix-Matrix Multiplication 96
References 99

1 MOTIVATION, HARDWARE TRENDS AND TECHNOLOGICAL CONSTRAINTS

The material presented in this chapter is an incomplete summary of the introductory chapter 1 of the "Parallel Computer Organization and Design" book [Dubois et al. 2012]. The hardware track of the PMPH course follows several chapters of the book, but these are not covered by lecture notes, for obvious (copyright) reasons. If the student would like to gain a deeper understanding of the hardware material, beyond what the *lecture slides* can offer, the course organizers recommend that the student acquires the book—it is a good one! The very simplified and adapted material presented in this section serves only as motivation for the software track of PMPH.

The over-arching motivation for the PMPH course is given by The Moore's Law[1] [Moore 2006]:

"The number of transistors in a dense integrated circuit doubles about every two years."

This law has been commonly rephrased as:

"Compute power doubles every 19*-to-*24 *months, while the cost effectiveness keeps pace."*

Cost effectiveness is typically expressed as the ratio between the performance and the cost of hardware, and "keeping pace" intuitively means that the increase in performance is free of charge, i.e., performance increases exponentially while the cost remains roughly the same.

Parallel architectures have been a very popular topic in the academic community, starting from early 80's—as demonstrated by a multitude of papers published in top conferences specialized in both hardware and software, such as the International Symposium on Computer Architecture (ISCA) and International Conference on Parallel Processing (ICPP). In essence, a large body of scientific work predicted (since the 80's) that the demise of single-CPU systems was inevitable and fast approaching. It took however almost two decades (mid 2000) until this prediction was ultimately validated in practice.

What happened in the meantime was the so called *"killer-micro"* effect: The additional hardware resources generated by the rapid increase in transistor density were utilized to increase the speed/frequency of the single-CPU systems. This resulted in complex designs of muscled (single) processors, relying on out-of-order (data-flow) execution model, that were capable of (i) storing in their pipelines thousands of instructions and (ii) executing hundreds of instructions per cycle.

The divergence between academia and industry was primarily motivated by a very pragmatic consideration: muscling the single-CPU architecture was seen as *the path of least resistance*, since it allowed all existent software to directly benefit from improvements without necessitating any modification/adjustment. In contrast, the transition to parallel architectures requires significant re-writing of the code base, which is not only tedious, but highly nontrivial: It requires reasoning about loop parallelism, and furthermore, it requires reverse-engineering a set of commonly-applied optimizations, tributary to the sequential-thinking era—e.g., related to memory and register savings—which significantly obfuscate the parallel semantics of the underlying algorithm.[2] Suffice to say that in that period (90's - mid 2000), multi-processor architectures were seen in the commercial arena only as exotic extensions of a single-processor architecture.

In 2004 Intel cancels the design of the Pentium4 @4GHz uniprocessor, which signals a tectonic shift towards multiprocessor design. From this point on, academia and industry are in agreement: all future architectures must adopt some form of massive parallelism in order to keep the Moore's Law alive. What prompted the shift to multiprocessor was that the uniprocessor design hit some walls (or reached a peak), beyond which it was impractical to scales this technology. Common

[1]Moore's Law is an observation and projection rooted in historical trends and does not constitute a physical or natural law.
[2]Of note, one of the mainstream programming languages of the time was Fortran77; many loops were implemented with jump instructions, and the RAM memory was in the range of kilobytes to several megabytes, which prompted aggressive memory reuse across loop iterations that obfuscate parallelism.

examples are the power wall—e.g., the dynamic power is proportional to the cube of frequency—and the memory wall—the seemingly exponentially-increasing performance gap between processor and memory.

The rest of this chapter will briefly look at the hardware trends related to the critical components of a parallel system—processor and memory—and will briefly review a set of important technological constraints—such as power, reliability, wire delays, design complexity. The intent is to demonstrate that *parallel*-hardware design fits well with the trends and addresses well the technological constraints.

In fact, nowadays, commodity architectures available mainstream (such as GPUs) provide thousands of cores and tens of thousands of hardware threads. What is problematic nowadays, is actually the lack of (high-level) programming models and compiler optimizations that would allow the development of commodity software to unleash the power of the already-available highly-parallel hardware. This would be the subject of the other chapters of the software track of PMPH.

1.1 Abstractions

We will use the following abstractions, which we define rather informally.

A program is a set of statement performaing computational tasks, while *a process/thread* embeds the execution of the computation. A good analogy is that a program is to process/thread what a recipe is for cooking.

A processor (core) is the hardware entity capable of sequencing and executing the process/thread's instructions.

Multi-threaded cores support multiple hardware threads, each running in its hardware context.

A multiprocessor is a set of processors connected to execute a workload. They are mass produced, off the shelf. Each multiprocessor consists of several cores—potentially with hardware multi-threaded support—and several levels of cache. The trend has been (and still is) to migrate system functions on the chip—e.g., memory controllers, external cache directories, network interface.

1.2 Processor Frequency and Number of Transistors

Figure 1 shows that historically, the clock rate (frequency) at which instructions are executed has increased exponentially between 1990 to 2004.

The bolded line in the figure depicts an uniform increase of 1.19× per year, the dotted line depicts an increase of 1.49× per year, and the continuous line marked with black rectangles depicts the actual increase in the clock rate.

The 1.19 exponential corresponds to technology scaling: the same hardware is being built on new technology. As silicon technology improves, the distances shrink (a.k.a., process shrinking). A new technology generation happens about once every two years, and in each generation transistors' switching speed increases about 41%, which directly translates to an increase in the clock rate.

Between the years 1990 – 2002 the actual increase in clock rate has matched the 1.49 exponential: clock rate has doubled every 21 months. If that trend would have continued, we would have had processors running at 30GHz by 2008!

The difference between the 1.49 and 1.19 exponentials (up until 2002) corresponds to improvements in the processor design. Examples include:

(1) Designing very-deep pipelines, consisting of 10 – 20 stages. Having more stages in the pipeline means that each stage is less complex and thus it requires a smaller number of gates per stage. This means that the execution of a stage is quicker, which allows to increase the clock rate. Historically, the number of gate delays has dropped by 25% every process generation.

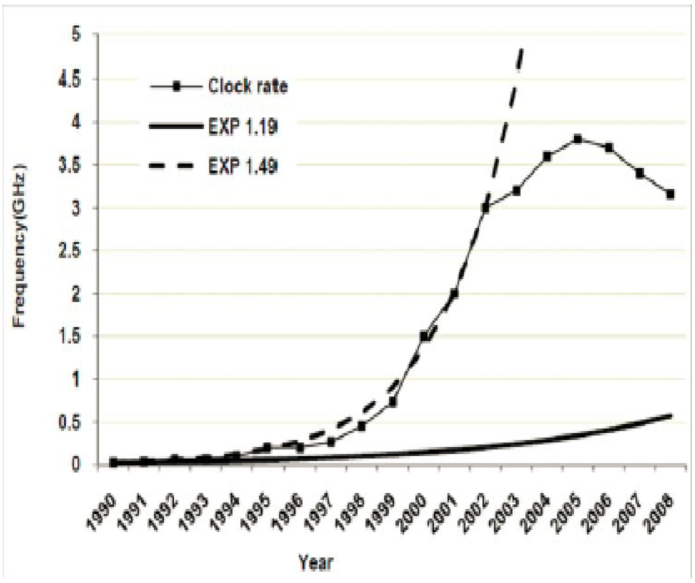


Fig. 1. Processor's Clock Frequency (Rate) between 1990-2008.

(2) aggressively exploiting instruction-level parallelism (ILP), for example by means of out-of-order, speculative execution, which combine techniques such as register renaming, reordering buffers, branch predication, lockup-free caches, speculative memory disambiguation.
(3) improvements in circuit design.

In 2004, Intel cancels the design of the Pentium4 @4Ghz, and switches track to multi-core design. This moment constitutes a tectonic shift away from the muscled deeply-pipelined uniprocessor. The clock rate peaked in 2005 but has mostly stalled since 2002.

In essence, further increase of the clock rate is unsustainable because of a number of reasons:

- First, it is unfeasible to build deeper pipelines because it is difficult to imagine useful stages that can be built from less than 10 gates (we have already reached that point).
- Second, the impact of technology scaling will be blunted in the future due to wire delays, because the speed of wire transmission grows much slower than the switching speed.
- Finally, and perhaps most importantly, circuits clocked at higher rates consume more power, and we have already reached the limits of power consumption in single-chip microprocessors.

However, it is still the case that each process generation—which roughly happens every two years—offers an additional budget of resources, such as transistors, that can be utilized to increase performance in other ways than sustaining clock-rate increases. Figure 2 shows historical data related to how fast the feature size—a unit proportional to the gate length—has shrunk over the years and how fast the number of transistors have grown. In essence, every two years we have a new process generation, in which the feature size is reduced by about 30% every generation. The number of transistors also seems to double every two years (according to Moore's law), reaching one billion in 2008.

The question thus becomes how to best utilize these hundreds of billion of transistors in the quest for ever higher performance. The design of highly-parallel hardware is one (if not the only) viable direction in this sense. For example the budget of transistors can be used to:

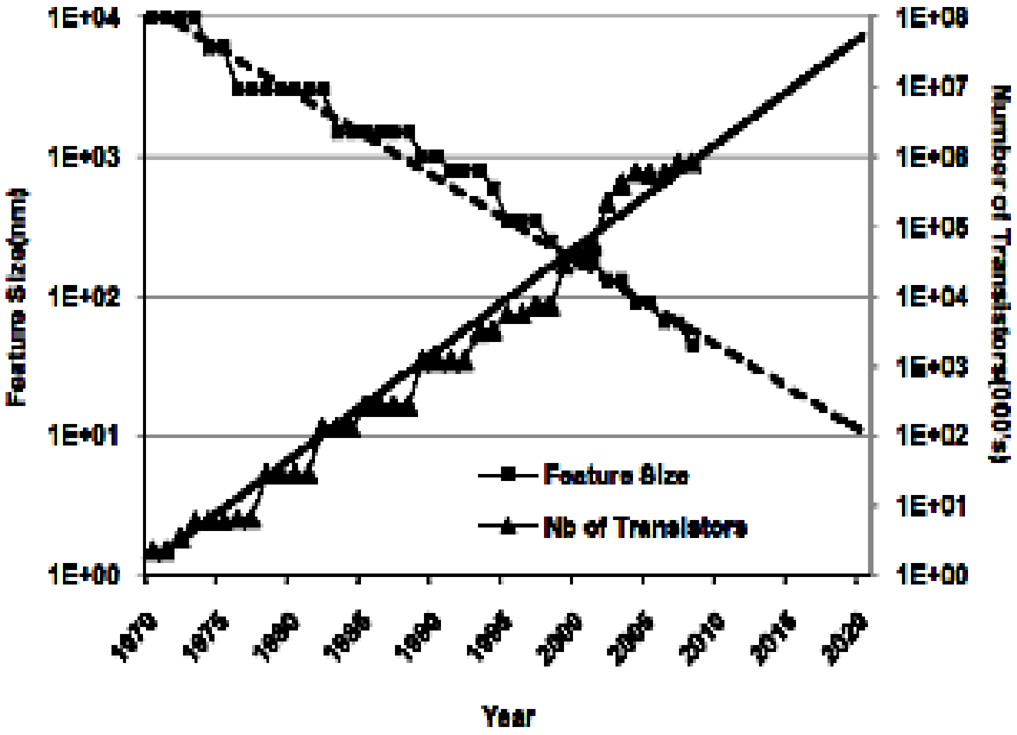


Fig. 2. Historical data and prediction related to the feature-size shrinkage and the number of transistors increase. The feature size is shown by the line starting in the top-left corner; the number of transistors is shown in the line starting in the bottom-left corner.

- enhance the parallelism of the memory system,
- to fetch and decode multiple instructions per clock,
- to run concurrently multiple hardware threads per core, for example in order to hide the (high) latency of the memory system,
- to support thousands of cores that run threads in parallel on different cores.

1.3 Memory Wall! Which Memory Wall?

The term "memory wall" was coined to denoted the seemingly ever-growing gap between the processor and memory speed. This wall is important because no matter how fast the processor is, it still needs to wait for the memory system to deliver the data to be processed.

The historical trends related to memory have been that DRAM density increases 4× every three years, but DRAM speed increases only 7% every year. This in comparison to the processor speed increasing for a long time by 50% per year.

Figure 3 shows the historical data related to the memory wall, which is defined as the ratio between the memory cycle and the processor cycle: In 1990, the memory wall was about 4, i.e., the processor was running at about 25MHz, while the memory cycle took about 150 nanoseconds. (Since 1Hz is 1 cycle/sec, then the processor cycle was about 40 nanoseconds.)

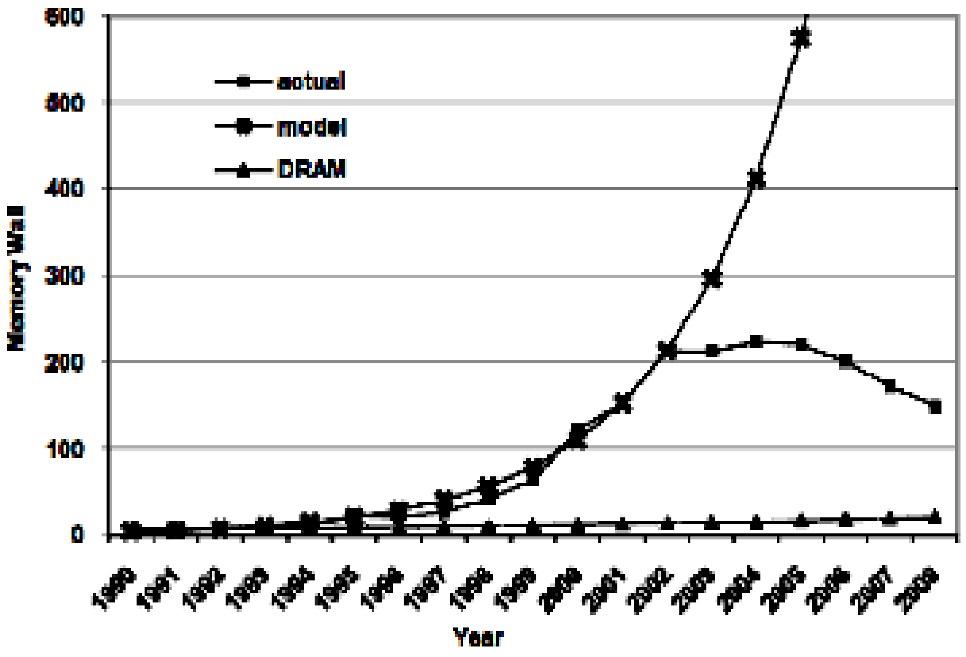


Fig. 3. Historical data related to the memory wall.

The memory wall has grown exponentially until 2002, when it reached 200, and the perception was that the memory wall was going to last/grow forever. However, it has stopped growing and actually has declined since 2004 because the clock rate of the uniprocessor could not be increased anymore, while the DRAM speed still grows, albeit slower.

As such, the advent of multi- and many-core systems have rendered the memory wall obsolete, but have introduced instead a bandwidth wall. This is because nowadays, the memory subsystem needs to efficiently feed cores that execute threads in parallel, which means that the memory system has to be capable of delivering multiple data in the same time, which is measured by bandwidth.

1.4 Technological Constraints

In the past, the main trade-off related to architecture design has been between cost (area) and time (performance). Today, the architectural design is challenged by several technological limits, such as power, wire delays, reliability, complexity of design. We will briefly examined each of them in the following (sub)sections, and would conclude that parallel architectures seem to address well all these constraints.

1.4.1 Power.

The major new constraint is power consumption, which is the sum of dynamic and static powers:

$$\texttt{Total Power} = P_{dynamic} + P_{static}$$

The dynamic power is consumed every time a gate is switching states, i.e., from 0 to 1 or from 1 to 0, hence it is mostly dissipated in processors. It can be computed by the formula:

$$P_{dynamic} \ = \ \alpha\, C\, V^2\, f$$

where V denotes the supply voltage, f denotes the clock rate, T denotes the temperature, and α denotes the activity factor (i.e., αf is the rate at which the gates switch). In a given circuit, an increase in frequency requires a proportional increase in the supply voltage as well, and a decrease in frequency allows similarly to reduce the supply voltage. It follows that the dynamic power consumed is roughly proportional to the cubic power of the frequency, i.e., $P_{dynamic} \sim f^3$. As such, dynamic power consumption clearly favors parallel processing over increasing the clock rate of the uniprocessor. For example, increasing the frequency by a factor of 4× consumes $4^3 = 64\times$ more dynamic power, but replicating a uniprocessor running at the original frequency 4 times consumes only 4× more dynamic power.

The static (leakage) power is dissipated in all circuits, at all times, no matter of frequency and whether the circuit switches or not. In practice, it is dominated by cache leakage. It can be computed with the formula:

$$P_{static} \ = \ V\, I_{sub} \ \sim \ V\, e^{-kV_T/T}$$

where V_T denotes the threshold voltage—the voltage at which a transistor switches off. One can observe that the leakage power increases exponentially as V_T is reduced and as T is increased.

The leakage power was negligible 15 years ago, but since the feature size and the threshold voltage decreases with every process generation, the leakage is getting worse. Currently, the leakage power has overtaken dynamic power as the major source of dissipation.

1.4.2 Reliability.

Hardware errors/failures can be classified into several categories:

- **Transient Failures (Soft Errors).** The charge stored in a transistor is $Q = C\ V$, where C is the capacitance and V is the supply voltage. In every process generation the supply voltage is reduced in order to maintain the electrical field at a constant strength. Thus Q is considerable reduced at every process generation, which results in every bit of storage in caches and processors being more prone to flip bits due to various corruption sources, such as cosmic rays, alpha particles radiating from the packaging material, electrical noise. In essence, the device is operational, but the data has been partly corrupted. To protect against such faults, DRAM/SRAM provide some form of error detection and correction capabilities.
- **Intermittent/Temporary Failures** occur due to environmental variations on the chip, such as high temperature (hot spots). In order for the device to return to correct behavior, the cause of the errors needs to be removed; for example the device should be switched off for a while, so that the temperature drops. It follows that temporary failures last longer than transient failures, but they still allow to continue execution (by temporarily switching off the faulty device).
- **Permanent Failures** result in permanent damage to the device, which will never function properly ever again, and thus the device must be isolated and replaced by a spare one.

Chip mutiprocessors promote better reliability than uniprocessor systems: For example, threads can be used to redundantly perform the same computation, and a voting mechanism can be employed to decide the correct answer, and to temporarily disable a core/resource. Similarly, faulty cores can be detected and disabled automatically, while the remaining system remains functional, albeit at a reduced capacity. This allows a natural failsafe degradation of the system.

1.4.3 Wire Delays.

Each process generation shrinks distances, thus enhancing miniaturization. The consequence is that transistors switch faster, but the propagation of signals on wire does not keep pace with this scaling.

To understand why this happens we take a look at the underlying physics. The propagation delay on a wire is proportional with the product between its resistance and capacitance $\sim RC$. The resistance, at its turn, is proportional with the ratio between the length and the cross-section area of the wire, i.e., $R \sim L/CS_{area}$. The length of the wire shrinks with every process generation due to miniaturization; that is good! The problem however is that the cross-section area shrinks as well due to the same reason, which annuls much of the length-shrinking benefits.

The impact of wire delays also favors multiprocessors, because communication traffic is hierarchical: most communication is local, while inter-core communication only happens occasionally.

1.4.4 Design Complexity.

Design verification has become the dominant cost of chip development today, and thus constitutes a major design constraint. The principal reason is that chip density increases much faster than the productivity of verification engineers. Much like in the case of software, this is due to the lack of new, high-level, productivity-oriented tools that also run fast. Verification is required at several levels of hardware design, such as:

- at the gate and register-transfer language level: verifying that the logic is correct,
- at the core level: verifying the correctness of forwarding and memory-disambiguation protocols,
- at multicore level: verifying the cache-coherency and memory-consistency protocols.

To some extent, these verification difficulties have resulted in dedicating the vast majority of chip resources to storage, simply because it is trivial to increase the size of caches, store/reorder buffers, load/store/fetch queues, etc., without compromising the safety of the design.

The design complexity trend also favors multiprocessors, as it is much easier to replicate the same structure multiple times than it is to design a large and complex system (or to bring improvements to one such). Similar to the case of storage, scaling up the number of cores of a multiprocessor should not raise major problems from a design perspective because, for example, any reasonable design of the cache-coherency infrastructure should be parametric in the number of cores.

1.4.5 CMOS Meets Quantum Physics.

CMOS[3] is rapidly reaching the limits of miniaturization: if the current trend continues, the feature size—defined as half the distance between two metal wires—will be less than 10 nanometers by year 2020. This means that the gate length, which is about half the feature size, would be in the range of 5 nanometers.

The radius of the atom is between 0.1 and 0.2 nanometers, and is not affected at all by the miniaturization trends. In essence, the gate length is quickly reaching the range of atomic distances, which are governed by quantum physics, where binary logic is replaced with probabilistic states.

While quantum computers are an area of active research, it is probably safe to say that commodity quantum hardware is not yet quite visible at the horizon. Until that time comes, the ever increase in compute power will be achieved by means of parallel architectures, such as (clusters of) many cores. And even if/when quantum hardware will emerge as a viable technology, it is also clear that quantum software will be massively parallel by nature, so that at least the principles of parallel programming will remain of interest.

[3]Complementary metal-oxide semiconductor, abbreviated as CMOS, is the (current) technology used for constructing integrating circuits.

2 LIST HOMOMORPHISM (LH)

The goal of the software track of PMPH is to teach students how to "think parallel". To achieve this goal we start by introducing in this section a very simple programming model, which utilizes only flat **`map-reduce`** operators, and which is rooted in the mathematical structure of list homomorphisms.

2.1 Math Preliminaries: Monoid and Homomorphism

This subsection briefly recalls the mathematical structures of monoid and homomorphism.

A monoid is a set tupled with an associative binary operator, which accepts an identity element within the set, and a group is a monoid in which any element is invertible. In formal notation:

Definition 1 (Monoid).
Assume a set S *and a binary operator* $\odot : S \times S \rightarrow S$.
$(S, \odot)$ is called a monoid *if it satisfies the following two axioms:*
(1) Associativity: $\forall x, y, z \in S$ *we have* $(x \odot y) \odot z \equiv x \odot (y \odot z)$ *and*
(2) Identity Element: $\exists e \in S$ *such that* $\forall a \in S$, $e \odot a \equiv a \odot e \equiv a$.

Definition 2 (Group).
$(S, \odot)$ *is called a group if it is a monoid satisfying the additional property that any element is invertible:* $\forall a,\ \exists a^{-1}$ *such that* $a \odot a^{-1} \equiv a^{-1} \odot a \equiv e$.

For example,

- $(\mathbb{Z}, +)$ denotes the monoid formed by the set of (signed) integers with the addition operation, which has 0 as neutral element. $(\mathbb{Z}, +)$ is also a group because any integer i has an inverse $-i$, which also belongs to $\mathbb{Z}$.
- $(\mathbb{N}, +)$ denotes the monoid of natural numbers with addition, which has 0 as neutral element. $(\mathbb{N}, +)$ is not a group because for example 1 does not have an inverse in $\mathbb{N}$.
- $(\mathbb{Z}, \times)$ is the monoid of integers with multiplication, which has 1 as neutral element. $(\mathbb{Z}, \times)$ is not a group because for example 2 is not invertible in $\mathbb{Z}$. ($\frac{1}{2} \notin \mathbb{Z}$)
- $(\mathbb{L}_T, ++)$, is the monoid formed by the set of lists of elements of some type T ($\mathbb{L}_T$), together with the list concatenation operator (++), which has the empty list ([]) as neutral element. $(\mathbb{L}_T, ++)$ is obviously not a group.

A monoid homomorphism is a function between two monoids, such that operations on one monoid can be directly mapped into operations on the second monoid. In formal notation:

Definition 3 (Monoid Homomorphism).
A monoid homomorphism *from monoid* $(S, \oplus)$ *to monoid* $(T, \odot)$ *is a function* $h : S \rightarrow T$ *such that* $\forall u, v \in S$, $h(u \oplus v) \equiv h(u) \odot h(v)$.

2.2 The Shape of a List-Homomorphic Function/Implementation

Throughout this chapter, we will work with finite lists, albeit the gained insight will carry over to arrays, which is the main datatype promoting efficient execution on modern hardware. We recall that $(\mathbb{L}_T, ++)$ is a monoid:

- ++ denotes list concatenation, for example
  `[1, 2, 3] ++ [4, 5, 6, 7] ≡ [1, 2, 3, 4, 5, 6, 7]`
- `[]` denotes the empty list, which is the neutral element for concatenation:
  ∀ list `x`, we have that `[] ++ x ≡ x ++ [] ≡ x`

We will use the term *list-homomorphic function* (LHF) to denote a function, which accepts an implementation (LHI) that uses a certain form of divide and conquer programming, as defined below.

Definition 4 (List-Homomorphic Function/Implementation LHF/LHI). *A LHF function h is a (mathematical) function that can be implemented as:*

```
h( [ ] )    = e
h( [x] )    = f(x)
h( x ++ y) = h(x) ⊙ h(y)
```

In simple words a LH implementation consists of:

(1) A divide-and-conquer case `h( x ++ y) = h(x) ⊙ h(y)`, which allows to partition the input list `z` into any two sublists `x` and `y` such that `z = x ++ y` and the implementation applies recursively `h` to the sublists and combines the results with the operator $\odot$. We draw attention to the following important observations:
   (a) in order for h to be well defined as a mathematical function (i.e., for the implementation to be useful in practice), h needs to compute the same result no matter of how the input list is partitioned into `x` and `y`. This is an implicit assumption.
   (b) `h( x ++ y) = h(x) ⊙ h(y)` essentially defines a homomorphism between the monoid $(\mathbb{L}_T, ++)$ and another monoid $(Img(h), \odot)$ whose set is the image[4] of `h` and its binary operator is $\odot$. One remaining question is: "who is the neutral element of $(Img(h), \odot)$?"

(2) Two base cases corresponding to `h` being applied to the empty list and to a list formed by exactly one element:
   (a) `e` is the implementation of the empty-list case, i.e., `h( [ ] ) = e`. It can be proven that `e` is actually the neutral element of $(Img(h), \odot)$.
   (b) the other base case is implemented as the application of a function `f` to the (single) element of the list.

The LH implementation is unsuitable to being mapped efficiently to modern highly-parallel hardware, for example because GPUs[5] do not support recursion. However, we remark that the function `f`, the binary operator $\odot$ and the neutral element `e` are explicitly provided by the LHI and the First Theorem of List Homomorphisms (explained later in section 2.4.1) allows to straightforwardly translate this LHI to a (map-reduce) implementation that is amenable to efficient GPU execution. The following theorem summarizes our previous observations:

Theorem 1 (LHF is a Mathematical Homomorphism). *A list-homomorphic function (which computes the same result regardless of how the list is partitioned):*

```
h( [ ] )    = e
h( [x] )    = f(x)
h( x ++ y) = h(x) ⊙ h(y)
```

is a mathematical homomorphism from $(\mathbb{L}_T, ++)$ to $(Img(h), \odot)$.
In particular $(Img(h), \odot)$ must be a monoid with neutral element `e`*, which also means that $\odot$ must be associative.*

The reverse also (trivially) holds: if `h` *is a homomorphism between $(\mathbb{L}_T, ++)$ and some monoid $(M, \odot)$ then it accepts a LHI (as above).*
The proof *is left as an exercise.*

[4]The image of a function $f : A \rightarrow B$ is the subset of B covered by the application of f to all elements of A.
[5]We use GPU as the abbreviation for general-purpose graphical processing units.

2.3 Examples of List-Homomorphic Implementations

We have discussed so far list homomorphism at a rather abstract level. This section aims at demonstrating that list-homomorphism programming is actually quite natural (when it fits) by examining several simple code examples. The examples use a notation that somewhat resembles Haskell, for example

- greeks such as α denote arbitrary types (type variable), and $[\alpha]$ denotes the type of a list whose elements are of type α,
- `f a b` applies the function `f` to two arguments `a` and `b`,
- &&, `||` correspond to logical and, or operators.

We discuss the following examples:

len: The first example corresponds to computing the length of a list, i.e., `len : [`α`] → Int`.

```
len [ ] = 0
len [x] = one x -- where one x = 1
len (x ++ y) = (len x) + (len y)
```

The first base case says that an empty list has size 0. The second base case says that a list containing exactly one element has size 1; however the LHI form requires to call a function on the list's single element, so we have defined the function one to always return 1 no matter the argument. Finally, the divide-and-conquer case says that if we partition the input list into two sublists then the length of the initial list is the sum of the lengths of the two sublists.

all$_p$: The second example corresponds to the function `all`$_p$ that checks whether some predicate `p :` α `→ Bool` satisfies (holds on) all elements of the input list:

```
allp [ ]      = true
allp [x]      = p x
allp (x ++ y) = (allp x) && (allp y)
```

The base cases say that an empty list satisfies the predicate (why?) and that a list containing exactly one element satisfies the predicate if and only if the element satisfies the predicate (obviously). The recursive, divide and conquer case says that a predicate satisfies a list (`z = x ++ y`) if and only if it it satisfies both (sub)partitions `x` and `y`. The implementation of the empty-list case must be **`true`** because this is the neutral element of the monoid formed by the two-element set {`true, false`} and the logical-and operator (&&). Further intuitive confirmation is given by the fact that if `all`$_p$ `[]` is chosen **`false`** then the function `all`$_p$ is ill-defined, in that it can generate two different results for the same input. For example, assume `p x = x > 3`. Then `all`$_p$ `[5, 6]` should result in **`true`** because all its elements are greater than three. However, the following legal derivation produces **`false`**:

`all`$_p$ `[5, 6] ≡ all`$_p$ `([] ++ [5, 6]) ≡ (all`$_p$ `[]) && (all`$_p$ `[5, 6]) ≡` **`false`** `&& (all`$_p$ `[5, 6]) ≡` **`false`**.

sum: The third example corresponds to summing up the (numerical) elements of a list. Without further explanation, this can be accomplished with the following LHI:

```
sum [ ]      = 0
sum [x]      = id x -- where id x = x
sum (x ++ y) = (sum x) + (sum y)
```

fld: The fourth example corresponds to the function that adds two to every (numeric) element of a list and then multiplies the results. In F# this can be expressed by a `fold` operation:

```
fold (\acc x -> acc * (x+2)) 1 mylist
```

Without further ado, this can be accomplished with the following LHI:

```
fld [ ]    = 1
fld [x]    = plus2 x -- where plus2 x = x + 2
fld (x ++ y) = (fld x) * (fld y)
```

2.4 Map, Reduce and List Homomorphism Theorems

This section (i) starts by introducing two of the (three) important basic blocks that lay the foundation of data-parallel programming (**map** and **reduce**), then (ii) presents the first theorem of list homomorphism, which basically gives a straightforward way of translating a LHI into a semantically-equivalent **map-reduce** implementation, and (iii) it concludes by presenting several other theorems that can be seen as simple re-write rules that enable various program optimizations.

It is perhaps important to notice that while the discussion refers to lists, this is only to preserve consistency with the list-homomorphism presentation. List is an implicitly sequential datatype; whenever we mention lists from now on, the reader should think arrays, at least in what parallel execution is concerned.

The first basic-block of data parallel programming is the **map** second-order operator, which takes as argument an unary function and a list of elements, and produces a list of the same length as the original one by applying the function argument to each element of the input list. The type and semantics of **map** are presented below:

map : $(\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta]$
map f $[x_1, \ldots x_n]$ = $[f\ x_1, \ldots, f\ x_n]$

Please note that, assuming a pure-functional language and replacing lists with arrays, **map** has implicit parallel semantics, because the computation corresponding to some element x_i of the array is completely independent of the computations of all other elements of the array.

The second basic-block of data parallel programming is the **reduce** second-order operator, which takes as arguments an *associative* binary operator, the neutral element corresponding to the monoid induced by the binary operator, and a list of elements. The result of **reduce** is obtained by successively applying the operator to all the elements of the array. The type and semantics of **reduce** are presented below:

reduce : $(\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow [\alpha] \rightarrow \alpha$
reduce $\odot$ e $[x_1, \ldots x_n]$ = $e \odot x_1 \odot \ldots \odot x_n$

Unlike **map**, the parallel semantics of **reduce** is not straightforward to see. We demonstrate it in fig. 4: assuming an infinite number of processors, the parallel computation of **reduce** is performed by means of a reduction tree, which performs sequentially a number of `log(n)` parallel operations, where `n` stands for the length of the input array – i.e., the computations on each breadth level in the tree are performed in parallel. It is perhaps important to stress (again) that $\odot$ **must be associative**:

- In parenthesis, that is why we did not bother to place the parenthesis that would specify the execution order, but the commonly-used sequential implementation of **reduce** accumulates to the left, i.e., $((e \odot x_1) \odot \ldots \odot x_n)$.
- More importantly, if $\odot$ is not associative then the sequential and parallel execution of **reduce** will likely give different results. For example, assume the list `[1,2,3,4]` which is reduced with the non-associative operator `acc` $\odot$ `x = acc + 2*x`. Even ignoring the neutral element, sequential execution will result in $(((1 \odot 2) \odot 3) \odot 4) = 19$ and parallel execution will

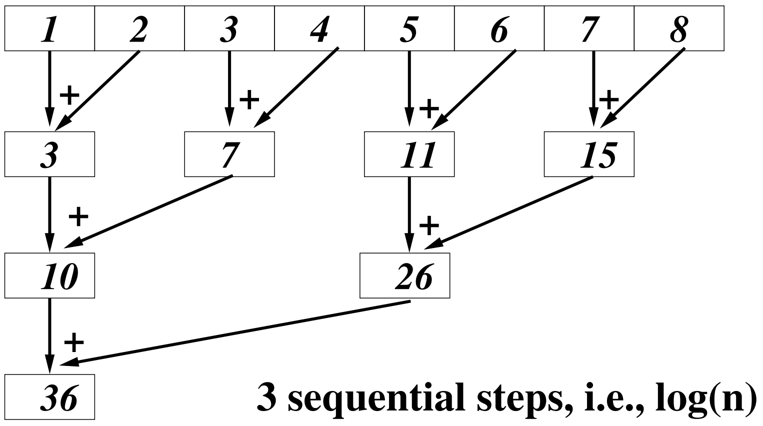


Fig. 4. Parallel execution of **reduce** requires a sequence of `log(n)` parallel operations (n is the array length).

result in `(1 ⊙ 2) ⊙ (3 ⊙ 4) = 5 ⊙ 11 = 27`, according to the execution patterns of the reduction tree shown in fig. 4.

- It should be clear now that **reduce** is different that **fold** (from F#). This can be seen from the type of `fold :` $(\beta \to \alpha \to \beta) \to \beta \to [\alpha] \to \beta$; in particular it does not make sense to talk about the associativity of **fold**'s operator because of the wrong type—associativity makes sense on binary operators whose arguments and result have the same type. Moreover, even when the type happens to be correct, as in the previously-discussed case, we have still seen that **fold** is not a parallel operator because it does not require associativity. In particular `fold (\acc x -> acc + 2*x) 0 lst` can be parallelized by re-writing it as a **map-reduce** composition: `reduce (+) 0 (map (+2) lst)`!
- In practice forgetting that the **reduce** operator must be associative—i.e., using **reduce** with non-associative operators—is one of the main generators of difficult-to-find bugs.

Similarly, remember that `e` must be the neutral element of the monoid defined by ⊙—hence reducing an empty list results in `e`, and `e` can be safely omitted from the computation if the list is not empty (because by the definition of the neutral element, we have that `e ⊙ x ≡ x, ∀ x`).

2.4.1 *First List-Homomorphism Theorem.*

The first LH theorem [Gibbons 1996] basically gives a straightforward way of rewriting a LHI as a **map-reduce** composition, as stated in the theorem below, which uses ∘ to denote the function-composition operator.

THEOREM 2 (1st LH THEOREM).
A list-homomorphic implementation defined as:
```
h( [ ] )    = e
h( [x] )    = f(x)
h( x ++ y) = h(x) ⊙ h(y)
```
is semantically equivalent with (`reduce ⊙ e`*)* ∘ *(*`map f`*), or in complete code:*
```
h z ≡ reduce ⊙ e (map f z)
```

Proof: *exercise.*

Theorem 2 also provides the theoretical argumentation of why the **reduce** operator must be associative: if h is a list homomorphism, then $(Img(h), \odot)$ must be a monoid, and by definition, a monoid requires its operator $\odot$ to be associative.

Applying theorem 2 to the LHI discussed in section 2.3 results in the following **map-reduce** implementations:

- `len z` ≡ **`reduce`** `(+) 0 (`**`map`** `one z)`
- `all`$_p$ `z` ≡ **`reduce`** `(&&)` **`true`** `(`**`map`** `p z)`
- `sum z` ≡ **`reduce`** `(+) 0 (`**`map`** `id z)` ≡ **`reduce`** `(+) 0 z`
- `fld z` ≡ **`reduce`** `(*) 1 (`**`map`** `plus2 z)`

We have thus started from an arguably natural program specification rooted in the mathematical theory of list homomorphisms, and we have translated that into an implementation in which parallelism is made explicit by **map-reduce** operators and which can be straightforwardly mapped to modern parallel architectures. We discuss simple program optimizations next.

2.4.2 Other List-Homomorphism Lemmas.

Theorem 3 (LH Promotion Lemmas).
Given unary functions `f` *and* `g`*, and an associative binary operator* $\odot$ *with neutral element* $e_\odot$ *then the following three identities hold, where* $\circ$ *denotes function composition:*

```
1. (map f) ∘ (map g) ≡ map (f ∘ g)
2. (map f) ∘ (reduce (++) []) ≡ (reduce (++) []) ∘ (map (map f))
3. (reduce (⊙) e⊙) ∘ (reduce (++) []) ≡
                              (reduce (⊙) e⊙) ∘ (map (reduce (⊙) e⊙))
```

If you are unfamiliar with the functional notation for composition, the identities can be written in full as:

```
1. map f (map g zs) ≡ map (λ z → f (g z)) zs
2. map f (reduce (++) [] zs) ≡ reduce (++) [] (map (λz→map f z) zs)
3. reduce (⊙) e⊙ (reduce (++) [] zs) ≡
                              reduce ⊙ e⊙ (map (λ z → reduce ⊙ e⊙ z) zs)
```

The identities can be seen as re-write rules that can be used to optimize the program in various ways, for example:

(1) The first identity is known as the **map** fusion/fission rule:
   ⇒ Fusion corresponds to applying the transformation in the forward (⇒) direction, and is useful for reducing the number of accesses to global memory, which is much slower than accessing the data stored in registers. For example, the left-hand side program: **`let`** `tmp =` **`map`** `g zs` **`in`** **`map`** `f tmp` requires to read and write in the first **map** each element of arrays `zs` and `tmp`, respectively, followed by reading and writing in the second **map** each element of `tmp` and result arrays, respectively. This counts up to 4 accesses to global memory per element. The right-hand side program: **`map`** `(λ z → f (g z)) zs` requires reading and writing the input and result arrays only once, thus halving the number of accesses to global memory (assuming that `f` and `g` operate on scalars and that the intermediate computation `(g z)` is held in registers, not in memory).
   ⇐ Fission corresponds to applying the transformation in the backward (⇐) direction and is useful for enhancing the degree of parallelism that is statically mapped to hardware, in

the context of a nested-parallel program. (This will be discussed in detail later on, in the context of the flattening transformation and vectorization.)

(2,3) Similar to fusion/fission, the second and third identities can be used in the forward direction ($\Rightarrow$) to efficiently sequentialize the parallelism in excess of what the hardware can support, and in the backward direction ($\Leftarrow$) to enhance load balancing and the program's degree of parallelism that can be statically mapped to hardware.

$\Rightarrow$ Assume `split`$_p$ denotes the operator that splits a list into p sublists of roughly equal lengths. Please observe that `(reduce (++) [])` $\circ$ `split`$_p$ $\equiv$ `id`, meaning that splitting a list into p sublists, then flattening the resulted list results in the original list. (`id x = x` stands for the identity function.) One may straightforwardly derive a new identity, presented below by composing both sides of identity (2) with `split`$_p$:

`map` `f` $\equiv$ `(`**`reduce`** `(++) [])` $\circ$ `(`**`map`** `(`**`map`** `f))` $\circ$ `split`$_p$ (2')

The difference is that (2) necessarily operates on lists, while (2') operates on list of lists. We are interested in using this new identity in the forward ($\Rightarrow$) direction. Assume the hardware has p cores, and that the input list (array) contains n elements, where n is much larger than p. The left-hand side **`map`** `f` suggests an execution model that spawns n threads—this is suboptimal on many architectures. The translation basically aims to spawn a number of threads equal to the number of cores. This is achieved by splitting the list, then processing sequentially each chunk on one core (by the inner **`map`**), while processing the p chunks in parallel (by the outer **`map`**), and finally by concatenating the per-core results. Similar thoughts apply to the third identity.

$\Leftarrow$ Consider a program similar to the left-hand side of the original identity 2. Its input is necessarily a list of lists. Assume the input list has p unbalanced sublists, for example all sublist have 2 elements, except for the last one which has $n - 2 \cdot p - 2$ elements, where n is big. If executed as suggested by the right-hand side—each core processes a sublist—the parallel execution will be utterly unbalanced because the last core will process many more items than the rest of the cores. If processing an item uniformly takes one unit, then the speedup achieved by the right-hand side program will be $\frac{n}{n-2 \cdot p-2}$ which converges to 1 when n goes to infinity. Instead, one can apply the second identity in the $\Leftarrow$ direction to flatten parallelism, then one can apply again the forward direction as in the $\Rightarrow$ bullet above by splitting the concatenated list again into p sublists of *roughly-equal* lengths. The execution of the resulting program is now load-balanced, each core processing a similar number of elements, resulting in a speedup close(er) to the optimal $p\times$. Similar thoughts apply to the third identity.

The final theorem is often used to optimizing the scheduling of **`map-reduce`** computations, for example in frameworks such as OpenMP. The idea is that a **`map-reduce`** composition can be re-written into a semantically equivalent program that:

- splits the input list into p sublists of roughly equal length,
- applies the original computation to each sublist, such the computation of a sublist is performed sequentially on a core, but different sublists are processed in parallel on different cores,
- applies the original reduction to the per-core results.

The benefit of such an execution is not only given by spawning a number of threads equal to the numbers of cores, thus reducing scheduling and switching-contexts overheads, but also optimizing the reduction depth. Originally, the reduction was applied to a list of n elements, and would require $log(n)$ sequential steps (see reduction tree in fig. 4). In the transformed program the final (parallel) reduction is performed on a list of p elements, requiring only $log(p)$ sequential steps.

Theorem 4 (Optimized Map-Reduce Lemma). *Assume* $\texttt{split}_p :: [\alpha] \rightarrow [[\alpha]]$ *distributes a list into p sublists, each containing about the same number of elements. Also assume* $\odot$ *a binary associative operator with neutral element* $\texttt{e}_\odot$ *and* `f` *a unary function. The following identity always holds:*
$\textbf{redomap}\ (\odot)\ \texttt{f}\ \texttt{e}_\odot \equiv$
$(\textbf{reduce}\ (\odot)\ \texttt{e}_\odot) \circ (\textbf{map}\ (\textbf{redomap}\ (\odot)\ \texttt{f}\ \texttt{e}_\odot)) \circ \textbf{split}_p$

where **redomap** *is defined as* $\textbf{redomap} \odot \texttt{f}\ \texttt{e}_\odot \equiv (\textbf{reduce} \odot \texttt{e}_\odot) \circ (\textbf{map}\ \texttt{f})$.
The Proof *is left as an exercise.*

In what the proof of theorem 4 is concerned, the big hint is to first observe that $(\textbf{reduce}\ (\texttt{++})\ [\,]) \circ \texttt{distr}_p$ results in the identity function, which is the neutral element for function composition—concatenating the result obtained by splitting an input list results in the input list. As such we can start by composing the left-hand side with the identity written as before:
$\textbf{redomap}\ (\odot)\ \texttt{f}\ \texttt{e}_\odot \equiv (\textbf{reduce}\ (\odot)\ \texttt{e}_\odot) \circ (\textbf{map}\ \ \texttt{f}) \circ (\textbf{reduce}\ (\texttt{++})\ [\,]) \circ \texttt{split}_p \equiv \ldots$
and then it takes about three applications of the promotion lemmas of theorem 3 to derive the right-hand side of the identity stated by theorem 4.

2.5 Almost/Near Homomorphisms [Gorlatch/Cole]

The notion of *near homomorphism* [Cole 1993] or synonymously *almost homomorphism* [Gorlatch 1996] has been introduced (independently) by Murray Cole and Sergei Gorlatch, respectively.

The simple intuition is that a non-homomorphic function g can be sometimes "lifted" into a homomorphic one, by computing a baggage of extra information. If this is possible, then the result of the original problem can be obtained by projecting the homomorphic result (e.g., by selecting an element from a tuple).

We will demonstrate the near-homomorphism construction on two interesting problems, which are going to be examined and solved in the rest of this chapter.

2.5.1 Maximum-Segment Sum (MSS) Problem. :

The formulation of MSS problem is:

"Given a list of signed integers, find the contiguous segment of the list whose members have the largest sum among all such segments; the result is only the maximal sum, not the segment's members."

For example, the MSS of `[1, -2, 3, 4, -1, 5, -6, 1]` is 11, and the corresponding maximal segment is `[3, 4, -1, 5]`.

One can observe that it is impossible to express this problem directly in a list-homomorphism way—such that the operator of the reduction receives two integers as arguments and produces an integer, i.e., $\odot$: `int` → `int` → `int`. For example, assume the list has been split as l_1 =`[1, -2, 3, 4]` and l_2 =`[-1, 5, -6, 1]`. According to the definition of MSS, the human can observe that the MSS of sublist l_1 is 7 (corresponds to segment `[3,4]`), and the MSS of sublist l_2 is 5 (corresponds to segment `[5]`).

Having the result of MSS for each sublist summarized only as an integer prevents us from meaningfully combining the sublists results into a result that is correct for the whole list. For example, with what operator should we combine 7 and 5? Should it be addition, which would result in MSS being 12, or should it be the maximal value, which will result in 7, or what?

Neither give the correct result, which we recall is 11 for the given input. The reason is that the maximal segment can very well lie across l_1 and l_2, i.e., partly in l_1 and partly in l_2, but this case cannot be covered by a reduce operator working on integers—we need to provide the reduce operator with additional information.

For example, one can reason that it would be useful to maintain for each (sub)list:

(mis) an integer corresponding to the maximal sum across all contiguous segments that starts the list (i.e., those containing the first element); we will name this the maximal-initial sum `mis`,

(mcs) and similar for the contiguous segments that ends a list (i.e., those containing the last element); we will name this the maximal-concluding sum `mcs`. With this extra information one could reason that the operator that combines the results of two sublists l_1 and l_2 should chose the maximal value between the MSS of l_1, the MSS of l_2, and the maximal segment that crosses l_1 and l_2 (i.e., lies partly in l_1 and partly in l_2), which is obtained by adding the $\texttt{mcs}_1$ of l_1 with the $\texttt{mis}_2$ of l_2. (Since we consider only contiguous segments, a crossing segment *will necessarily be* a composition of an `mcs` of l_1 with an `mis` of l_2.) It would seem that we have successfully figured out how to compute the MSS of two sublists, but this computation requires the `mis` and `mcs` of the two sublists; how do we compute those?

(ts) To compute `mis` and `mcs` we need only one extra piece of information: the total sum of a sublist, denoted as `ts`. Assume we have the results for l_1 and l_2 and we want to compute the `mis` for $l = l_1{+}{+}l_2$. We can reason that the `mis` of l is the maximal value between:
- the $\texttt{mis}_1$ of l_1, because the initial segments of l_1 are also initial segments of l, and
- $\texttt{ts}_1 + \texttt{mis}_2$, because the maximal initial segment of l may span across its two sublists—in this case, by definition of initial segment, it necessarily needs to include the whole sublist l_1 and the maximal initial segment of l_2.

Similar considerations apply to computing the `mcs` of of l.

We have applied above a list-homomorphic (divide-and-conquer) type of reasoning, in that we have derived what the reduce operator should be in terms of thinking how to combine the results of two sublists into the result of a list. We are now ready to write directly the **`map-reduce`** (obtained by applying theorem 2). The Futhark implementation is given below; **`i32`** denotes the 32-bit signed-integer type, (**`i32`**,**`i32`**,**`i32`**,**`i32`**) is the type of a quad-tuple of **`i32`**s, and (1**`i32`**, 2**`i32`**, 3**`i32`**, 4**`i32`**) is a quad-tuple value of that type:

```
let mssOp (mss₁: i32, mis₁: i32, mcs₁: i32, ts₁: i32)
          (mss₂: i32, mis₂: i32, mcs₂: i32, ts₂: i32)
        : (i32,i32,i32,i32) =
    let mss = i32.max (i32.max mss₁ mss₂) (mcs₁+mis₂)
    let mis = i32.max mis₁ (ts₁ + mis₂)
    let mcs = i32.max mcs₂ (ts₂ + mcs₁)
    let ts  = ts₁ + ts₂
    in  (mss, mis, mcs, ts)

let f (x: i32) : (i32, i32, i32, i32) =
    (i32.max x 0, i32.max x 0, i32.max x 0, x)

let projectFst (x, _, _, _) = x

let maxSgmSum (xs: []i32) : i32 =
    let exp_xs  = map f xs
    let exp_res = reduce mssOp (0,0,0,0) exp_xs
    in  projectFst exp_res
```

In essence, the implementation of MSS, denoted `maxSgmSum` has three main steps:

map: first each element of the input array `xs` is lifted to a quad-tuple, by applying `f`—this allows to compute a larger baggage of information, i.e., `mis, mcs, ts`;
reduce: then the result is reduced with the operator ⊙ as explained above;
project: finally, we select (project) the first element of the result tuple that contains the information of interest (the MSS) and discard the rest.

The implementation above diverges a bit from the definition of MSS, in that the result is always positive—we kept this form in order to be consistent with the original paper [Cole 1993]. If we would like also to compute negative MSS values, we can change the implementation of `f` to **`let`** `f x = (x,x,x,x)` and the neutral element of ⊙ to $(-\inf, -\inf, -\inf, 0)$.

2.5.2 *Longest-Satisfying Segment (LSS) Problem.* :

LSS denote a class of near-homomorphic problems which requires to find the length of the longest (contiguous) segment of a list for which some property holds. For example, we might want to compute:

zeros: the length of the longest segment of zeros, or
same: the length of the longest segment made from the same number, or
sorted: the length of the longest sorted sequence.

Please notice however, that it is *not* the case that all predicates result in a LSS problem that can be expressed as a list (near) homomorphism. For example the length of the longest sequence whose sum is 0 is *not* expressible as a list homomorphism.

It turns our that if we restrict the predicate to have a certain shape, then all such predicates allow a list-homomorphic implementation. The shape of the restricted predicate is:

```
p []          = true
p [x]         = ... -- some implementation
p [x, y]      = ... -- some implementation
p (x:y:zs) = (p [x,y]) && (p (y:zs))
-- where && denotes the logical-and operator
-- and : denotes the cons operator, i.e., x : y : z : [] == [x,y,z]
```

Note that various predicates can be implemented by filling in the computation of the base cases when the list contains one and two elements, respectively. For example, the predicates for the three list-homomorphic problems listed above are derived by the following implementation of the base cases:

```
zeros [x]   = (x == 0)
zeros [x,y] = (zeros [x]) && (zeros [y])

same [x]   = true
same [x,y] = (x == y)

sorted [x]   = true
sorted [x,y] = (x <= y)
```

Now that we finally defined what a LSS problem is, it remains to reason about what should be the baggage of extra information that would allow us to lift a LSS problem to accept a list-homomorphic implementation. The rational is somewhat similar to the one for maximal segment sum, but with several additions:

- As before, we need to maintain the *length* of the longest initial and concluding satisfying segments, which we denote by `lis` and `lcs`, respectively, together with the total length of the (sub)list, denoted by `tl`.
- When considering the concatenation of the ($\texttt{lcs}_1$, $\texttt{lis}_2$) pair—i.e., for the case when the segment of interest spans across the two sublists—it is not guaranteed that the spanning segment satisfies the predicate; this must be explicitly checked! For example, if the elements of some lists x and y are in sorted order, this does not means the list obtained by concatenating x with y has elements in sorted order: (sorted l_1) && (sorted l_2) $\not\Rightarrow$ sorted (l_1++l_2).
- To perform the check mentioned above, we need to record the *last* element of `lcs` and the *first* element of `lis`. This would allow to compute whether $\texttt{lcs}_1$ is *connected* to $\texttt{lis}_2$, by checking the condition `p [lastx,firsty] == True`.

2.5.3 *Exercise: Longest-Satisfying Segment (LSS) Implementation.* :

The implementation of the longest-satisfying segment is proposed as an exercise (first weekly assignment). You will need to fill in the blanks in the skeleton code below. You should also implement it in Futhark, run it on the GPU-equipped machines and report the speedup between the accelerated and CPU-based versions (obtained by compiling with `futhark opencl` and `futhark c`, respectively):

```
let redOp (lss₁:i32, lis₁:i32, lcs₁:i32, tl₁:i32, frst₁:i32, last₁:i32)
          (lss₂:i32, lis₂:i32, lcs₂:i32, tl₂:i32, frst₂:i32, last₂:i32)
        : (i32, i32, i32, i32, i32, i32) =
    let connect = ... -- fill in the blanks
    let lss     = ... -- fill in the blanks
    let lis     = ... -- fill in the blanks
    let lcs     = ... -- fill in the blanks
    let tl      = ... -- fill in the blanks
    let frst    = if tl₁ == 0 then frst₂ else frst₁
    let last    = if tl₂ == 0 then last₁ else last₂
    in  (lss, lis, lcs, tl, frst, last)

let mapOp (x: i32) =
    let xmatch = if (p [x]) then 1i32 else 0i32
    in  (xmatch, xmatch, xmatch, 1i32, x, x)

let projectFst (a, _, _, _, _, _) = a

let lgstSatSgm (xs : []i32) : i32 =
    let exp_xs  = map f xs
    let exp_res = reduce (⊙) (0,0,0,0,0,0) exp_xs
    in  projectFst exp_res
```

2.6 Conclusion

This section has started by presenting a program specification rooted in the mathematical structure of list homomorphism, and which corresponds to an (arguably) natural, divide-and-conquer way of expressing a class of simple problems. We have then shown that such a program has an implicitly parallel semantics because it can be straightforwardly translated into a semantically-equivalent program written in terms of **`map-reduce`** compositions.

We have drawn attention that the reduce operator must be associative because the homomorphism theory requires it, and we have also drawn attention that a `fold` does not always have parallel semantics, because not even that its semantics does not require its binary operator to be associative, but we cannot even talk about it's operator's associativity (wrong type). However, we have seen that frequently, a `fold` can be rewritten by means of a **`map-reduce`** composition.

Next, we have studied several (promotions) lemmas derived naturally from the list-homomorphism theory, and we have explained that they can be seen as re-write rules, and can be used to optimize a program in different ways.

Finally, we have examined two classes of problems (MSS and LSS), whose (efficient) parallel nature is far from trivial even to understand, and we have shown that list-homomorphic (divide-and-conquer) reasoning can straightforwardly derive efficient parallel implementations for these problems.

The next section 3 will introduce several other parallel operators that are basic blocks of data-parallel programming, such as **`scan`**, and will demonstrate how to reason about the efficiency of a parallel program in terms of asymptotic properties such as work and depth. More importantly, it will show that complex programs can be built as puzzles from a *nested* composition of operators such as **`map`**, **`reduce`**, **`scan`**.

Furthermore, section 4 will give the intuition behind a transformation that can automatically rewrite a nested-parallel program into a flat-parallel one that can be statically mapped to highly-parallel hardware, in a way that preserves the asymptotic work-depth properties of the original (nested-parallel) program.

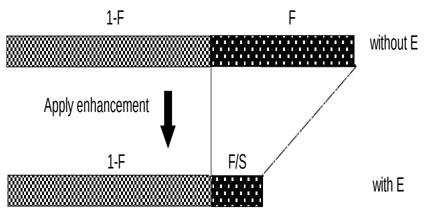


Fig. 5. Enhancement E accelerates a fraction F of a program by a factor S.

3 WORK-DEPTH ASYMPTOTIC, NESTED PARALLELISM

This section is organized as follows:

- section 3.1 introduces and demonstrates how to characterize parallel programs in terms of work and depth complexity.
- section 3.2 extends the set of parallel operators—with scan, filter, scatter—and demonstrates how several parallel programs, such as sparse matrix-vector multiplication, prime number computation, quicksort—can be elegantly expressed by nesting parallel constructs, and how their efficiency can be reasoned in terms of their work-depth asymptotic.

3.1 Reasoning in Terms of Work-Depth Asymptotic

This section starts by presenting Amdahl's law as a way to motivate why it is necessary to write programs as if the hardware provides an infinite number of cores. We then briefly present a simplified and idealized parallel hardware (PRAM) that is used to introduce the notion of work and depth complexity of a (nested) parallel program.

We then argue, by means of Brent's theorem, that the work-depth measure is a good approximation of the parallel behavior of the program, and we demonstrate how one can reason about computing the program's work and the depth for the simple case of summing up the elements of an array.

3.1.1 Amdahl's Law.

Figure 5 shows a scenario in which an "enhancement" accelerates the computation of a fraction F of a program on a fixed dataset[6] by a factor of S, while the other part of the program $1 - F$ does not benefit from it. (It is not important what the enhancement actually is, or whether it is of software or hardware nature.)

In this scenario, the execution time of the enhanced program is:

$$T_{exe}(withE) = T_{exe}(withoutE) \times [(1-F) + \frac{F}{S}]$$

Amdahl's Law correspond to (the interpretation of) three formulas: one that computes the speedup of the enhanced program:

$$Speedup(E) = \frac{T_{exe}(withoutE)}{T_{exe}(withE)} = \frac{1}{(1-F)+\frac{F}{S}}$$

and another two that compute an asymptotically-tight upper bound for the speedup when S goes to infinity:

$$Speedup(E) \leq \frac{1}{1-F} \text{ and } \lim_{S\to\infty} Speedup(E) = \frac{1}{1-F}$$

[6]In general, a program running time is sensitive to the dataset, so it does not makes sense to talk about speeding up a fraction F of a program in general—we need to fix the dataset.

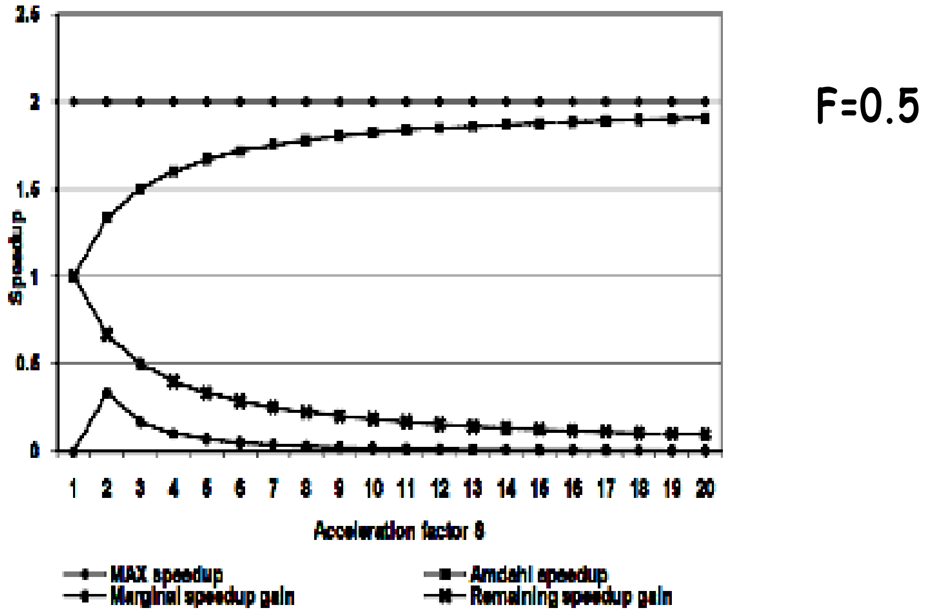


Fig. 6. Interpretation of Amdahl's Law by Diminishing Returns. The x and y axis show the acceleration factor S and the speedup, respectively. The lines appearing in the figure from top to bottom are: the maximal permitted speedup, the Amdahl's speedup, the remaining speedup gain, the marginal speedup gain (between two consecutive values of S).

In essence, Amdahl's law shows that no matter how big the improvement is, the overall application speedup is limited by the $1 - F$ fraction that does not benefit from the improvement.

Figure 6 shows a more detailed interpretation of Amdahl's Law, specialized for the value $F = 0.5$, hence the maximal speedup is $2\times$. The figure demonstrates the law of diminishing returns. It is reasonable to assume that every increment of S—shown on the x axis—requires the same amount of additional resources (for enhancement). However, every increment of S is less and less rewarding globally. For example the step from $S = 2$ to $S = 3$ generates an overall program speedup of 33%, while moving from $S = 5$ to $S = 6$ generates a much smaller increase in speedup of only 6.67%. The moral of the story is to realize that some games cannot be won—the program speedup will never be higher than $2\times$—so one should know when to stop, i.e., at the point when the next speedup gain will not justify the (extra) cost of the resources necessary for implementing the next unit of enhancement. In what hardware design is concerned, this would mean to implement the "common case[7]" in hardware (hence fast), and execute the rare case in software (e.g., exceptions).

Figure 7 depicts the interpretation for applying the Amdahl's law to the particular case of parallelism:

$$Speedup(P) = \frac{T_1}{T_P} = \frac{P}{F+P(1-F)} < \frac{1}{1-F}$$

The specialization is straightforward: utilizing P cores (rather than one) ideally results in a $P\times$ speedup. Actually, that is not quite true, since *the occasional* super-linear speedup may be observed due to cache effects—the P cores together have $P\times$ more cache than the uniprocessor, and

[7]The common case is typically determined by benchmarking.

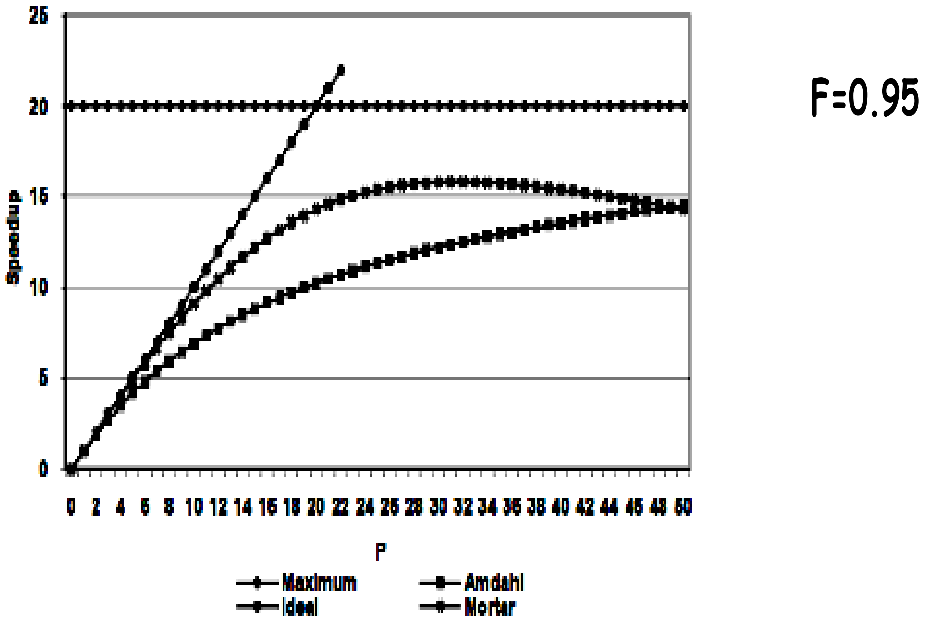


Fig. 7. Demonstrating Amdahl's Law when the enhancement is parallel execution and $F = 0.95$. The mortar line shows a "typical" evolution of speedup in practice.

applications with regular access patterns may benefit for the cache increase. But typically, the speedup is sublinear even if $F = 1$, because for example threads might need to communicate and communication is expensive. One can observe that the Amdahl's law is unforgiving: the figure uses $F = 95$, which corresponds to 95% of the runtime being run in parallel; still the speedup is limited to 20×, no matter how many cores you throw at it.

The moral of the parallel case is different than the one for the general hardware improvement. We do not advocate to bound/restrict the number of cores just because some applications are inherently sequential and will not benefit for extra parallelism. Quite the contrary: after all, we have seen that scaling hardware parallelism is the only conceivably way (nowadays) of keeping the Moore's Law alive.

What we advocate is to never leave sequential any part of the program that can possibly be parallelized. *In other words, when developing parallel code, we must reason as if the hardware has an unlimited/infinity number of cores.*

Let me stress this further with an example that may catch your attention: assume the student has parallelized 99.9% percent of the runtime of the application subject to the group project/exam. The student may feel entitled to receive the maximal grade, but the teacher might argue otherwise. The reason is that the GPUs that you are using in the PMPH course currently have about 7000 cores each. Assume for the sake of the argument that each core run as fast as the CPU (they do not!). Applying Amdahl's law for $F = 99.9\%$ results in a limiting 1000× speedup. This means that the student is only utilizing about one seventh of the compute power provided by one GPU, and 14% is a failing grade :))

3.1.2 *Work-Depth Asymptotic Behavior of a Parallel Program.*

While there is a trend towards simplifying hardware, current hardware remains way too complex for developing cost models based on them. At least in what teaching is concerned, using a realistic hardware model will put us in danger of “missing the forest for the trees”. We will discuss parallel-program properties by reasoning about the program execution time on a very simple and idealized hardware model, named the parallel random access machine (PRAM). PRAM focuses on (data) parallelism and ignores issues related to synchronization and communication. It assumes that:

- there are P processors that are connected to shared memory,
- each processor has an unique identifier/index $0 \leq i < P$,
- the execution happens in single-instruction multiple data (SIMD) fashion, which means that all cores execute in lock step—i.e., a core cannot start the next instructions until all cores have completed executing the current instruction. Please note that in the case of an **`if-then-else`**, the processors that did not take the **`then`** branch must wait until all the other processors has finished executing the **`then`** branch, before starting to execute the **`else`** branch, and similar for the **`else`** branch.
- each parallel instruction takes unit time (the same amount of time no matter whether it is a simple or complex arithmetic operation or a memory access).
- each processor has a flag that controls whether it is active in the execution of an instruction (for example in order to implement **`if-then-else`**). If the processor is not active, then its noop does not count towards the work complexity (but it counts towards the depth complexity, because it is part of a SIMD computational step).

We are ready to define the work-depth asymptotic behavior of a parallel program on a PRAM machine. The work and depth computation assumes an infinity number of processors ($P = \infty$).

- **The work complexity** is the total number of operations performed to execute the program, i.e., the sum across all processors. We denote work by $W(n)$, where n is related to the size of the dataset/workload.
- **The depth or step complexity**, denoted by $D(n)$ is the number of sequential steps needed to execute the program.
- **A parallel implementation is work efficient** if its work complexity is asymptotically equal to that of the best sequential implementation of the same algorithm.

If we know (have computed) the work and depth of an implementation, then Brent’s theorem specifies “good” complexity bounds for a PRAM that has a finite number of P cores, where by good we mean tight enough to be useful (for reasoning) in practice.

Theorem 5 (Brent’s Theorem).
A parallel implementation that has depth $D(n)$ and work $W(n)$ can be simulated on a P-processor PRAM in time complexity T such that:

$$\frac{W(n)}{P} \leq T \leq \frac{W(n)}{P} + D(n)$$

3.1.3 *Demonstrating Work-Depth Computation for Reduction.*

The reduction tree for summing up (**`reduce`** `(+) 0`) the elements of an array of 8 elements is shown (again) in fig. 8. It is intuitively easy to generalize the work and depth for an array of n elements, which is summed up using $\frac{n}{2}$ processors:

- the work is $W(n) = n$, hence parallel summation is work efficient because the best sequential algorithm still performs $O(n)$ steps;

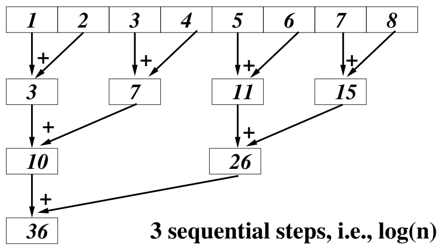


Fig. 8. Parallel execution of **reduce** requires a sequence of log(n) parallel operations (n is the array length).

```
Input:  array A of n=$2^k$ elems of type T
        $\oplus : T \rightarrow T \rightarrow T$ associative
Output: S = $\oplus_{j=1}^{n} a_j$

1.  forall i = 0 to n-1 do
2.    B[i] ← A[i]
3.  endfor

4.  for h = 1 to k do
5.    forall i = 0 to n-1 by $2^h$ do
6.      B[i] ← B[i] ⊕ B[i+$2^{h-1}$]
7.    endfor
8.  endfor
9.  S ← B[0]
```

Fig. 9. Imperative, low-level pseudocode for parallel array summation.

- the depth is $D(n) = log(n)$, i.e., number of sequential steps;
- the optimized runtime on P processors is actually $O((n/P) + log(P))$. This can be achieved by transforming the program according to theorem 4. We recall that the "Optimized-Map-Reduce Lemma" says that a **map-reduce** composition can be executed by:
  (1) chunking the input array into P subarrays of roughly-equal sizes,
  (2) then processing each subarray *sequentially* on a different processor, but *in parallel* across subarrays,
  (3) then reducing the results of the P subarrays in parallel by means of a reduction tree.
  Step (1) and step (2) are fully parallel and take $O(n/P)$ time, while step (3) requires a reduction tree on P elements, which takes $O(lg\ P)$ time (depth).
- one can now verify that Brent's lemma:

$$O(\tfrac{n}{P}) \;\leq\; O(\tfrac{n}{P} + log(P)) \;\leq\; O(\tfrac{n}{P} + log(n))$$

  gives bounds which are good approximations (for the assumed $P \leq \frac{n}{2}$) and allows us to reason on the essence rather than overthink the impact of optimizations.

We have so far derived the work-depth complexity of array summations in an intuitive way, by abstracting out the information depicted in a picture. The question is: "Can we also do this in a systematical way for an arbitrary code?" It turns out the answer is positive, as demonstrated by the following analysis of the low-level (imperative) pseudocode presented in fig. 9. The pseudocode uses **for** and **forall** to denote a sequential and a parallel loop, respectively. The analysis proceeds bottom-up, by which we mean that we go in-order across constructs at the same level and innermost-to-outermost in nests:

- The parallel loop between lines $1-3$ has $D_{1-3}(n) = \Theta(1)$, and $W_{1-3}(n) = \Theta(n)$,
- The parallel loop between lines $5-7$ has $D_{5-7}(n) = \Theta(1)$, and $W_{5-7}(n,h) = \Theta(n/2^h)$,
- The sequential loop between lines $4-8$ executes k iterations ($n = 2^k$, hence $k = lg\ n$), each consisting of the parallel loop $5-7$, hence it has:
  - depth $D_{4-8}(n) = k \times D_{5-7}(n) = \Theta(lg\ n)$, and
  - work $W_{4-8}(n) = \sum_{h=1}^{k} W_{5-7}(n,h) = \Theta(\sum_{h=1}^{k}(\frac{n}{2^h})) = \Theta(2n(1-\frac{1}{2^{k+1}})) = \Theta(2n(1-\frac{1}{2n})) = \Theta(n)$
- The statement on line 9 trivially has $D_9(n) = \Theta(1)$, $W_9(n) = \Theta(1)$,

- Thus the depth and work for the entire program are $D(n) = \Theta(lg\ n)$, and $W(n) = \Theta(n)$, respectively!
- By Brent's Theorem, it follows that the actual runtime is bounded by: $\frac{n}{P} \leq Runtime \leq \frac{n}{P} + lg\ n$

3.1.4 Naive and Native Implementation of Reduction in Futhark.

The previous section has shown that a reduction can be easily implemented based on **map**s and sequential loops. It is natural to ask then, why does reduction need to be a first-class citizen of a data-parallel language, when it can be easily be provided as part of a library?

We first provide an intuitive demonstration by implementing reduction in the Futhark data-parallel language and comparing the performance of our program with the natively supported reduction. This also allows us to get acquainted with Futhark, which will be used in PMPH exercises and so on.

The Futhark implementation of the reduction pseudocode discussed in the previous section is presented in fig. 10. In the following, we will explain the code:

(1) In Futhark comments start with token `-` and expand until the end of the line. However, an uninterrupted sequence of comment lines that start a source file (after the `-- ==` comment) define reference input-output datasets for automatic testing or benchmarking:
  - The first dataset is directly specified: the input consists of an array of 16 single-precision floats (**f32**), using the common array literal notation `[a`$_1$`, ..., a`$_n$`]`, and the reference result is a single precision float `7.0f32`.
  - The second dataset is held in a file (`-- compiled input @ data/f32-arr-16K.in`), that stores an array of length 16777216, but the result is given explicitly, albeit it can similarly be stored in a file (e.g., `-- output @ data/f32-arr-16K.out`).
  - The third dataset is an array of length 33554432 containing randomly generated single-precision floats. `auto output` means that the result is validated against the sequential-C execution; hence if the results differ then this possibly signals that a compiler bug exists.
  - Assuming the name of the file is `red-by-hand.fut`, compilation for the Opencl and Cuda GPU backends can be carried out with the commands `$ futhark opencl red-by-hand.fut` and `$ futhark cuda red-by-hand.fut` and compilation for sequential C can be performed with `$ futhark c red-by-hand.fut`; all will result in an executable named `red-by-hand`, which can be run on a dataset with the command:
    ```
    $ ./red-by-hand -e naiveRed -t /dev/stderr -r 10 < data/f32-arr-16K.in
    ```
    The `-t` option records the program runtime (in microseconds) in the next-specified file; in our case this is displayed in `/dev/stderr`. The reported runtime **does not include**:
    * the CPU-to-GPU transfer of the program input and the GPU-to-CPU transfer of the program result.

```
-- This implementation should be in a file named: red-by-hand.fut
-- ==
-- compiled input {
--   [ 1.0f32,-2.0f32,-2.0f32, 0.0f32, 0.0f32, 0.0f32, 0.0f32, 0.0f32
--   , 3.0f32, 4.0f32,-6.0f32, 1.0f32, 2.0f32,-3.0f32, 7.0f32, 2.0f32
--   ]
-- }
-- output {
--    7.0f32
-- }
--
-- compiled input @ data/f32-arr-16K.in
-- output { -948.970459f32 }
--
-- compiled random input { [33554432]f32 } auto output

-- For simplicity, assumes that n is 2^k
entry naiveRed [n] (a : [n]f32) : f32 =
  let k = i64.f32 <| f32.log2 <| f32.i64 n
  let b =
    loop b = a for h < k do
        let n' = n >> (h+1)
        in  map (\i -> #[unsafe] (b[2*i]+b[2*i+1]) ) (iota n')
  in b[0]

entry futharkRed [n] (a : [n]f32) : f32 =
  reduce (+) 0.0f32 a
```

Fig. 10. Futhark Implementation for Summing Up an Array of Single-Precision Floats

∗ the OpenCL/Cuda context creation and kernel compilation.
Option `-r 10` specifies that the runtime should be averaged across 10 runs. If one wishes to inspect various profiling information for the GPU backend, the program can be run with options `-L -P`; this is especially useful when program compilation generates a number of kernels.

– Testing and benchmarking across the datasets specified in the source file itself can be carried out with the commands:
`$ futhark test -backend=opencl red-by-hand.fut`, or
`$ futhark bench -backend=opencl red-by-hand.fut`, or
`$ futhark bench -backend=cuda red-by-hand.fut`

– random (uniformly-distributed) datasets can be created with the `futhark dataset` command (see help with the `-help` option). A dataset consisting of an array of 16777216 **f32** elements can be produced and saved in file `data/f32-arr-16K.in` by using the command:
`futhark dataset -b -f32-bounds=-1.0:1.0 -g [16777216]f32 > data/f32-arr-16K.in`

(2) Program entry points are specified with keyword `entry` and a command-line argument needs to specify which entry point is chosen for execution, e.g., `-e naiveRed`. If no entry-point

is specified, by default program execution will try to run the entry point called `main` (if one exists). For simplicity, try to not use tuples as arguments for entry points. A function declaration (which is not an entry point) starts with keyword **`let`**. The declaration of a function or entry-point includes its name, which is optionally followed by declaring a set of variables that will be used to denote array sizes, for example `[n][m][p]`, followed by a sequence of (optionally typed) formal arguments, and (optionally) by the return type of the function.

- Our `naiveRed` has one formal parameter, denoted `a` which is an unidimensional array of single-precision floats of length `n`, i.e., (`a: [n]f32`).
- A three-dimensional array in which the sizes of the first and last dimension are `n` and the size of the second dimension is `m` would be written as `[n][m][n]f32`, and `m` would need to be declared similar to `n`, i.e., `[n][m]`.
- In our case the result is a scalar: `f32`. The programmer has to be a bit careful with the sizes of an array result type: a size which is declared in the optional part can be used in the array-result type if and only if it has been used in one of the formal array arguments. For example, `let main [n] (a : [n]f32) : [n]f32 = ...` is legal and says that the input and result array should have the same length, but `let main [n] (m : f32) : [n]f32 = ...` is illegal because `n` cannot be deduced from the inputs. In the latter case, what the user intends is probably `let main (n : i32) (m : i32) : [n]f32 = ...`.

(3) For most cases of interest to us, the program input and result should be specified in tuple of arrays form (or structure of arrays AoS). (Note that this restriction only refers to the entry points; all other code may use array-of-tuple form.) For example, try not to pass to `main` a formal argument `b : [n](f32,f32)` or `c : (f32, f32)`; these should be split into two arguments each: `(b1: [n]f32) (b2: [n]f32)` or `(c1: f32) (c2: f32)`.

(4) The body of the `naiveRed` is a **`let`** expression.

1. `k` is bound to the value of `log2 n`:
   `let k = i64.f32 <| f32.log2 <| f32.i64 n`
   where `n` is assumed to be a power of two, and `<|` pipes the right-hand side result to the left-hand side function (call). **`i64.f32`** transforms a float argument into an 64-bit signed integer, and **`f32.i64`** performs the reverse operation.
2. `b` is defined to be the result of the **`loop`** expression: `loop b = a for h < k do body`, which is always executed sequentially:
   - ∗ `b` is a variable bounded in the loop context, whose value is variant across different iterations of the loop: it is initialized upon loop entry with the value of `a`, and the result of the loop body will give the value of `b` to be used by the next iteration of the loop.
   - ∗ the loop runs `k` iterations (we recall that `n = `2^k) and `h` takes values in `0, ..., k-1` in different iterations. Loops may also iterate across elements of an array using the more direct notation `for x in xs`, such that `x = xs[i]` in some iteration `i`.
3. The body of the loop consists of a **`let`** expression:
   - ∗ `let n ' = n >> (h+1)` which binds `n’` to the value obtained by bit-shifting `n` in the right direction with `h+1` bits, i.e., $n' = \frac{n}{2^{h+1}}$.
   - ∗ and results in the application of **`map`** second-order array combinator (SOAC):
     `map (\i -> (b[2*i]+b[2*i+1]) ) (iota n')`
     - · the array input of the **`map`** is `iota n'` which corresponds to the array `[0,1,...,n’-1]`.
     - · the mapped anonymous (lambda) function is `(\i -> b[2*i] + b[2*i+1])`.

     Please note that the Futhark implementation differs from the imperative pseudocode shown in fig. 9 in that each iteration of the Futhark loop computes an array result of half the size of the input b: iterations h=0, h=1 and h=k-1 result in arrays of length $n' = \frac{n}{2^1}$, $n' = \frac{n}{2^2}$ and $n' = \frac{n}{2^k} = \frac{n}{n} = 1$, respectively. That is the reason why all iterations use **map** with the same function (`\i -> b[2*i]+b[2*i+1]`).

4. Finally the loop execution results in an array b containing one element, and the `main` function returns the (first) element of b (i.e., `b[0]`).

The second entry point, named `futharkRed`, simply calls Futhark's native **reduce** construct.

Compiling and running the two programs on an A100 Nvidia GPU shows that the native reduce is about a factor 3× faster than our by-hand implementation (`red-by-hand.fut`) on dataset `data/f32-arr-16K.in`.

The reason for the speedup is that the code-generation of the native construct benefits from an optimized code generation that executes *one* kernel[8] that performs about n reads from global memory and much fewer writes to global memory, while using internally scratchpad (fast) GPU memory. In comparison, the discussed implementation (`red-by-hand.fut`) executes a $log(n)$-iteration loop, in which each loop iteration calls a kernel corresponding to the **map** of size $n' = \frac{n}{2^{h+1}}$. The **map** reads $2 \cdot n'$ and writes n' elements from/to global memory. In total this implementation performs about $2 \cdot n$ reads and n writes from/to global memory, which is about 3× more global-memory accesses than the native implementation.

This is one of the reasons for which it makes sense to have second-order array combinators (SOACs) such as **reduce** and **scan** as first-class citizens in the data-parallel language. The other reason corresponds to the algebraic properties of such operators: for example the composition of a reduce and a map can be fused in a more advanced construct [Henriksen et al. 2016; Larsen and Henriksen 2017], by a rule similar to theorem 4 which, similarly requires to read the input array once from global memory. Such high-level fusion rules would not be possible if we would implement the `reduce` in the library based on iterative applications of **map**; it is even difficult to see how fusion could be applied in such a context.

3.2 Other Parallel Operators, Examples of Nested-Parallel Applications

This section is organized as follows:

- section 3.2.1 introduces the type and semantics of several parallel operators commonly used in (functional) data-parallel languages.
- section 3.2.2 presents a possible, work-efficient implementation of the **scan** (a.k.a. parallel-prefix sum) operator, which is a basic-block of parallel programming, and shows how a segmented scan operator can be easily written in terms of **scan**.
- section 3.2.3 shows a possible implementation of **filter** based on **map** and **scan** and **scatter** (parallel write) operators.
- The remaining (sub)sections demonstrate how several applications can be constructed as puzzles from a nested composition of such operators:
  – section 3.2.4 briefly discusses the multiplication of a sparse matrix with a dense vector, and in particular introduces the notion of **data flattening**.
  – section 3.2.5 presents three implementations for computing all prime numbers less then a certain input: the first version provides the intuition, but does not have the optimal depth; the second version fixes depth optimality, but this does not means it is necessarily best in practice; the last version is work inefficient but has a structure that makes it efficient for GPU hardware.
  – Finally, section 3.2.6 looks at the implementation of quicksort.

[8]The code generation of `reduce` actually runs two kernels, but the second one takes negligible time.

```
map   : (α → β) → Πn. [n]α → [n]β
map   f [a₁, ..., aₙ] = [f a₁, ..., f aₙ]
map2 : (α1 → α₂ → β) → Πn. [n]α₁ → [n]α₂ → [n]β
map2 f [a₁, ..., aₙ] [b₁, ..., bₙ] = [f a₁ b₁, ..., f aₙ b₂]
map3 : ...

reduce : (α → α → α) → α →  Πn. [n]α → α
reduce ⊕ 0⊕  [a₁, ..., aₙ] = 0⊕ ⊕ a₁ ⊕ ... ⊕ aₙ

scan : (α → α → α) → α → Πn. [n]α → [n]α
scan ⊕ 0⊕ [a₁, ..., aₙ] = [a₁, a₁⊕a₂, ..., a₁ ⊕ ... ⊕ aₙ]

filter : (α → Bool) → Πn. [n]α → [m]α   (where m ≤ n)
filter p [a₁, ..., aₙ] = [a_k₁,..., a_kₘ]
  such that k₁ < k₂ < ... < kₘ, and denoting by k̄ = {k₁,..., kₘ},
  we have (p aⱼ == true) ∀ j ∈ k̄, and (p aⱼ == false) ∀ j ∉ k̄
```

Fig. 11. Types and Semantics of **map**s, **reduce**, **scan**, **filter**.

3.2.1 *Types and Semantics of Various Parallel Operators.*

We start the discussion with **zip** and **unzip** operators: **zip** receives two arrays of the same length n and produces an array of tuples of length n by pairing-up the elements at the same index of the two arrays. **unzip** is the inverse of **zip**, i.e., **unzip**=**zip**$^{-1}$:

```
zip : Πn. [n]α₁ → [n]α₂ → [n](α₁, α₂)
zip [a₁,..., aₙ] [b₁,..., bₙ] = [(a₁,b₁),...,(aₙ,bₙ)]

unzip : Πn. [n](α₁, α₂) → ([n]α₁, [n]α₂)
unzip [(a₁,b₁),...,(aₙ,bₙ)] = [a₁,..., aₙ] [b₁,..., bₙ]
```

Please note that our semantics differs from the one in Haskell, which would allow two list of different length and would result in an array of tuples whose length is the minimum of the input-array lengths. In Futhark **zip/unzip** are syntactic sugar: they are supported in the source language, but are eliminated (compiled away) in an early compiler stage that systematically rewrites the program to a tuple-of-arrays form.

We use the following array constructors:

```
iota : (n: i32) → [n]i32
iota n = [0,..., n-1]

replicate : (n: i32) → α → [n]α
replicate n a = [a,..., a]
```

iota n creates an arrays of integral elements starting from 0 to n-1, and **replicate** n a creates an array of length n filled with the same element a. **iota** can be used to create the iteration space, for example in the case when the parallel operator needs to access several elements of the input array in each iteration; we have seen such an example in the case of the naive-reduce implementation: `map (\i -> b[2*i] + b[2*i+1] ) (iota n')`. **replicate** is typically used for initializing an array which will be subject to a scan or to in-place updates.

Figure 11 shows the types and semantics of several second-order array combinators (SOACs). We have already introduce **map** and **reduce** combinators. For ease of notation we will also introduce **map2** which receives as arguments a binary function `f` and two input arrays of the same length, and produces an array by applying `f` to corresponding elements from the first and second arrays, respectively. Similar definitions are possible for `map3` and so on.

scan is commonly known as parallel-prefix sum, since it produces an array of the same length as the input by starting with the first element, then applying the operator between the first two elements, then applying the operator between the first three elements, and so on, until the last element corresponds to the result of reducing the array. Please note that, similar to **reduce**, **scan**'s binary operator $\odot$ must be associative! Please also note that the provided version of **scan** is *inclusive* and it does not actually require a neutral element; an *exclusive* **scan** requires a neutral element because it produces an array that starts with the neutral element and ends with the reduction of the first `n-1` elements. The work and depth of **scan** is also similar to **reduce**: $O(n)$ and $O(lg\ n)$, respectively.

Finally, **filter** receives as argument a predicate `p` (i.e., a function of type α →`Bool`) and an input array, and it filters-out from the input array all the elements that do not succeed under the predicate (i.e., indexes `j` such that `p` a_j `=` **false**). Please note that (i) the result elements respect the relative order in which they appear in the input array, and that (ii) the size of the result array is necessarily less than or equal to that of the input array. The work and depth of **filter** is similar to **scan** ($O(n)$ and $O(lg\ n)$), because its implementation uses **scan**.

We conclude this subsection by introducing **scatter**, a very important operator used for updating in parallel some of the indices of an arrays, where the updated indices do not necessarily follow any regular pattern, i.e., think parallel random writes. Its type is:
scatter: Π n. *[n]α → Π m. [m]**i64** → [m]α → *[n]α
which means that it receives as arguments a base array of length `n`, and two arrays of length `m` holding the to-be-updated indices and corresponding values, respectively, and produces an array of size `n` by applying the updates. We make the following important observations:

(1) The star * in front of the type of the first input array denotes an "unique" type, i.e., it specifies that the array is going to be consumed—since the update is performed in place, it means that any following reference to that array is *illegal*!
(2) Furthermore, a unique argument cannot "alias" any of the non-unique arguments, and similarly an unique result cannot alias any of the non-unique arguments (but can alias an unique input). This would not be necessary on a PRAM machine with an infinity number of cores that execute in SIMD fashion, but such hardware does not exist in practice. (For example, the result of **scatter** "aliases" the first input, since the update is performed in place, but not the other two input arrays.)
(3) The depth of **scatter** is $O(1)$, and its work is $O(m)$—meaning, it does not depend on `n`, the size of the to-be-updated array.

For programming convenience (e.g., padding), we enrich the semantics of **scatter** by requiring that the indices that are outside the bounds of the input array are simply ignored (are not updated). As such `n` and `m` are in no relation with each other: it can be that `n<m` or `n==m` or `n>m`.

We conclude with an example that hopefully clarifies how **scatter** works:

```
X (input  array) = [a₀, a₁, a₂, a₃, a₄, a₅]
I (index vector) = [2, 4, -1, 1]
D (data  vector) = [b₀, b₁, b₂, b₃]
scatter X I D    = [a₀, b₃, b₀, a₃, b₁, a₅]
```

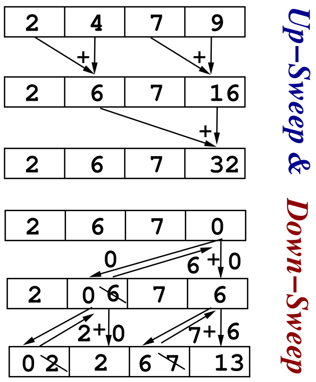


Fig. 12. Parallel execution of **scan** for a 4-element array.

3.2.2 *Implementation of Scan and Segmented Scan.*

We start by emphasizing again that, similar to **reduce**, **scan** requires an **associative** binary operator, and the exclusive scan requires the neutral element of the monoid induced by that operator.

The intuition behind the implementation of exclusive scan is depicted in fig. 12 for an array of four elements. Note that its semantics differs from that of Futhark's (inclusive) scan: assuming an n-element input array, an exclusive scan results in the neutral element in the first position of the result array, and the "sum" of the first n-1 elements in position n-1. The implementation is organized in two parallel steps:

(1) The first step is called "Up-Sweep" and is similar with a reduction, except that the accumulation of all elements is computed in the last element of the array, rather than the first.
  - After the up-sweep pass, value 0 is placed in the position of the last element.

(2) The second step is called "Down-Sweep", and it propagates updates to the array's elements in the reverse order of the up-sweep pass (i.e., reverse the arrows and the traversal of the up sweep). Each propagation requires two substeps:
  2.1. the left child sends its value to its parent and updates its value to that of the parent.
  2.2. the right-child value is obtained by applying the binary operator of the scan to the left-child value and to the (old) value of parent. Please notice that the right child is in fact the parent—an in-place algorithm.

The imperative pseudocode that implements the exclusive scan operator is shown in fig. 13. A reasoning similar to the one we applied to **reduce** can compute that the depth and work of the presented implementation is $D(n) = \Theta(lg\ n)$ and $W(n) = \Theta(n)$, respectively. In fact the only difference in comparison to the imperative pseudocode of reduce is that scan requires an extra (down-sweep pass), but this does not matter complexity-wise because it has the same work and depth as the up-sweep pass (similar to the one used for reduction), and a 2× factor leaves unchanged the asymptotic behavior.

The pattern of a work-inefficient algorithm for inclusive scan is shown in fig. 14, and the pseudocode is presented in fig. 15. This is the typical CUDA implementation for a warp of threads—a warp is the unit of parallel execution in CUDA and consists of 32 consecutive threads. Note that the depth of the implementation is optimal: $D(n) = \Theta(lg\ n)$, but the work is not: $W(n) = \Theta(n\ lg\ n)$. However, on CUDA platforms, any warp of threads executes in lock-step (in SIMD fashion), and de-selecting threads from execution (within one warp) brings no benefits. In fact this implementation

```
Input:  array A of n=$2^k$ elements of type $\alpha$
        $\oplus$ : $\alpha \rightarrow \alpha \rightarrow \alpha$ associative
Output: B = [0, $a_1$, $a_1 \oplus a_2$,...,$\oplus_{j=1}^{n-1}$ $a_j$]

1.  forall i = 0 : n-1 do
2.    B[i] ← A[i]
3.  endfor

4.  for d = 0 to k-1 do -- up-sweep pass
5.    forall i = 0 to n-1 by $2^{d+1}$ do
6.      B[i+$2^{d+1}$-1] ← B[i+$2^d$-1] ⊕ B[i+$2^{d+1}$-1]
7.    endfor
8.  endfor
9.  B[n-1] = 0
10. for d = k-1 downto 0 do -- down-sweep pass
11.   forall i = 0 to n-1 by $2^{d+1}$ do
12.     tmp ← B[i+$2^d$-1]
13.     B[i+$2^d$-1] ← B[i+$2^{d+1}$-1]
14.     B[i+$2^{d+1}$-1] ← tmp ⊕ B[i+$2^{d+1}$-1]
15.   endfor
16. endfor
```

Fig. 13. Imperative Pseudocode for implementing exclusive scan

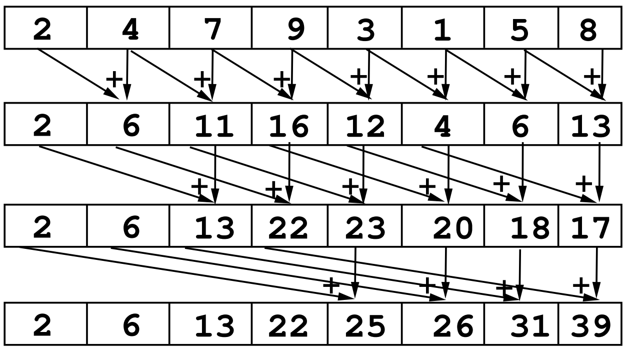


Fig. 14. Inclusive **scan** used inside a warp for CUDA implementation.

is a factor of 2× faster than the one based on the up- and down-sweep, because it performs only one sweep—its depth is *lg n* instead of 2 *lg n*.

The remaining of this (sub)section discusses the ***segmented scan*** operator. A segmented scan semantically operates on an array of arrays—think a matrix in which the rows do not necessarily have the same length—and it results in an array of arrays of similar shape as the input, in which each of the resulted subarrays are obtained by scanning the corresponding input subarray with the given associative binary operator (and neutral element). Thus the semantics of a segmented scan is

```
Input:  array A of n=$2^k$ elements of type $\alpha$
        $\oplus$ : $\alpha \rightarrow \alpha \rightarrow \alpha$ associative
Output: B = [$a_1$, $a_1 \oplus a_2$,...,$\oplus_{j=0}^{n-1}$ $a_j$]
1.  forall i = 0 : n-1 do
2.    B[i] ← A[i]
3.  endfor
4.  for d = 0 to k-1 do
5.    h = $2^d$
6.    forall i = h to n-1 do
7.      B[i] ← B[i-h] $\oplus$ B[i]
8.    endfor
9.  endfor
```

Fig. 15. Imperative Pseudocode for warp-level inclusive scan in CUDA; $n = 32$

a map over the input array, in which the mapped function performs a scan on each subarray. The example below demonstrates the semantics of an inclusive segmented scan:

```
sgmScan (+) 0 [[1,3,5], [7,8], [9,11,14,15]] ≡
[scan (+) 0 [1,3,5], scan (+) 0 [7,8], scan (+) 0 [9,11,14,15]] ≡
[[1,4,9], [7,15], [9,20,34,49]]
```

The example above specifies the semantics but does not give insight into what a data-parallel implementation should be. The major obstacle is that the array of arrays is typically represented as an array of pointers, each pointing to the corresponding subarray. However this representation is not suitable for parallel execution: we need a flat data-structure! As such, we represent the array of arrays by a flat array of *values*—whose length `n` is the total number of elements, i.e., the sum of the lengths of the subarrays—together with a *flag* array, which has `1` (or **`true`**) in the first position that starts a subarray, and `0` (or **`false`**) in the remaining positions. One may also add to the representation a shape array, which has number-of-subarrays integral elements, each containing the lengths of its corresponding subarray. We present an example below that demonstrates the data-parallel representation (shape + flag + value flat arrays):

```
nestedArray = [[1, 3, 5], [7, 8], [9, 11, 14, 15]]
   ↓             ↓          ↓        ↓
shapeArray  = [ 3,          2,       4              ]
flagArray   = [ 1, 0, 0,    1, 0,    1, 0,   0, 0   ]
valueArray  = [ 1, 3, 5,    7, 8,    9, 11, 14, 15  ]
```

It follows that the segmented scan has almost the same type as scan, the only difference being that it receives one extra argument: the flag array, which is represented as an array of booleans (or integers):

```
sgmScan : ($\alpha \rightarrow \alpha \rightarrow \alpha$) → $\alpha$ → Π n. [n]bool → [n]$\alpha$ → [n]$\alpha$
```

For completeness, fig. 16 shows a graphical representation of the execution pattern of exclusive segmented scan, and fig. 17 shows the imperative pseudocode for exclusive segmented scan. While there are more branches, it is relatively straightforward to see that the depth and work asymptotics of segmented scan remains the same as the one of scan: $D(n) = \Theta(lg\ n)$ and $W(n) = \Theta(n)$.

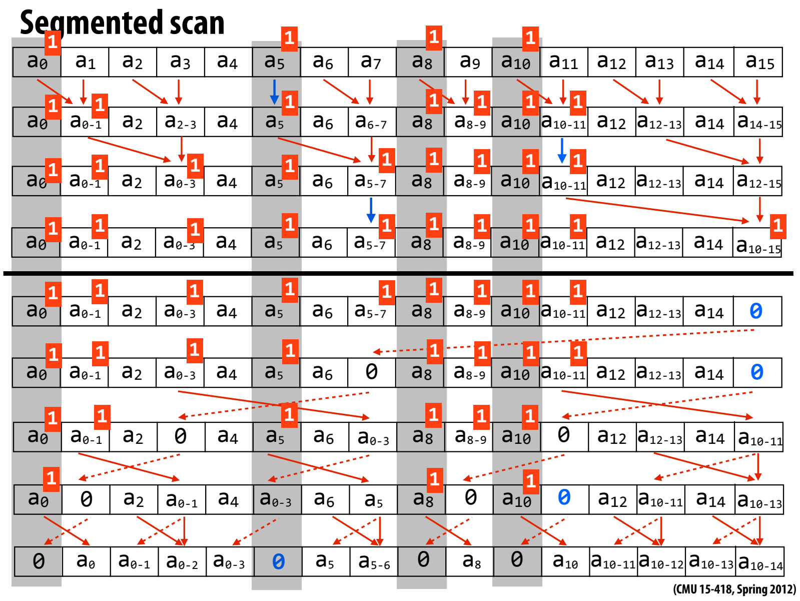


Fig. 16. Parallel execution of exclusive segmented scan. Figure courtesy of CMU 15-418, Spring 2012

Understanding the execution pattern and pseudocode is difficult: the teacher advices to *not* attempt it because it does not really provide essential new insight.

Instead, we make the ***essential observation*** that a segmented scan can be straightforwardly implemented in terms of a scan, which basically means that if an efficient implementation of scan is given, then we can directly derive an efficient implementation of segmented scan, and moreover, that segmented scan has the same work and depth complexity as scan.

The code below shows the Futhark implementation of the inclusive segmented scan:

```
let segmented_scan [n] 't (op: t -> t -> t) (ne: t)
                          (flags: [n]bool) (arr: [n]t) : [n]t =
  let (_, res) = unzip <|
    scan (\(x_flag,x) (y_flag,y) -> -- extended binop is denoted ⊙
             let fl = x_flag || y_flag
             let vl = if y_flag then y else op x y
             in  (fl, vl)
         ) (false, ne) (zip flags arr)
  in  res
```

The implementation consists of a **scan** whose input array is obtained by zipping the flag and value arrays, and whose binary associative operator combines two flag-value tuples:

- `(x_flag,x)` is the accumulator and `(y_flag,y)` corresponds to the flag and value of the current element of the input array;
- the segmented-scan operator computes the resulting value by checking whether the current element corresponds to the start of a segment (i.e., tests whether `y_flag` is true).

```
Input:  flag array F of n=$2^k$ of ints/bools
        data array A of n=$2^k$ elements of type $\alpha$
        $\oplus$ : $\alpha \to \alpha \to \alpha$ associative
Output: B = segmented scan of 2-dimensional (irregular) array A
1.  forall i = 0 to n-1 do B[i] ← A[i] endfor
2.  for d = 0 to k-1 do -- up-sweep pass
3.    forall i = 0 to n-1 by $2^{d+1}$ do
4.      if F[i+$2^{d+1}$-1] == 0 then
5.          B[i+$2^{d+1}$-1] ← B[i+$2^d$-1] $\oplus$ B[i+$2^{d+1}$-1]
6.      endif
7.      F[i+$2^{d+1}$-1] ← F[i+$2^d$-1] .|. F[i+$2^{d+1}$-1] -- .|. is bitwise-or
8.  endfor endfor
9.  B[n-1] ← 0
10. for d = k-1 downto 0 do -- down-sweep pass
11.   forall i = 0 to n-1 by $2^{d+1}$ do
12.     tmp ← B[i+$2^d$-1]
13.     if F_original[i+$2^d$] ≠ 0 then
14.         B[i+$2^{d+1}$-1] ← 0
15.     else if F[i+$2^d$-1] ≠ 0 then
16.         B[i+$2^{d+1}$-1] ← tmp
17.     else B[i+$2^{d+1}$-1] ← tmp $\oplus$ B[i+$2^{d+1}$-1]
18.     endif
19.     F[i+$2^{d+1}$-1] ← 0
20. endfor endfor
```

Fig. 17. Imperative Pseudocode for implementing exclusive segmented scan

  - if this is the start of a segment, then the resulting value is the current element value (y), as dictated by the semantics of inclusive segmented scan;
  - otherwise we need to accumulate the current value: `op x y`
- the segmented-scan operator computes the resulting flag by taking the logical or of the two argument flags. This is necessary in order to propagate the start-of-a-segment flag, because parallel execution may proceed in a different order than the sequential execution.
- We leave as an exercise to verify that the scan's (lifted) operator, named ⊙, is associative, i.e., `((x_flag,x) ⊙ (y_flag,y)) ⊙ (z_flag,z)` equals `(x_flag,x) ⊙ ((y_flag,y) ⊙ (z_flag,z))`, and to check that the neutral element of the monoid induced by ⊙ is indeed `(`**`false`**`, ne)`.

An important observation is that segmented scan can be easily adapted to work with an integral array of flags, in which a flag different than 0 denotes the start of a new segment. The necessary modifications are:

- `fl = x_flag || y_flag` is rewritten as `fl = x_flag | y_flag`, where | denotes the bitwise-or operator, and
- the branch condition `if y_flag` is rewritten as `if y_flag != 0`.

Maintaining the flags as an array of boolean reduces memory footprint and saves bandwidth, but some of the flattening rules can be optimized if we use the integral representation.

```
let partition2 [n] 't                      -- Assume t = i32, n = 6,
           (p : (t -> bool))               -- p (x:i32)= 0 == (x%2),
           (arr : [n]t) : ([n]t, i64) =    -- arr = [5,4,2,3,7,8]
  let cs  = map p arr                      -- cs  = [F,T,T,F,F,T]
  let tfs = map (\f -> if f then 1         -- tfs = [0,1,1,0,0,1]
                          else 0) cs
  let isT = scan (+) 0 tfs                 -- isT = [0,1,2,2,2,3]
  let i   = isT[n-1]                       -- i   = 3

  let ffs = map (\f->if f then 0
                        else 1) cs         -- ffs = [1,0,0,1,1,0]
  let isF = map (+i) <| scan (+) 0 ffs     -- isF = [4,4,4,5,6,6]
  let inds= map3 (\c iT iF ->              -- inds= [3,0,1,4,5,2]
                    if c then iT-1
                         else iF-1
                 ) cs isT isF
  let r = scatter (copy arr) inds arr      -- r = [4,2,8,5,3,7]
  in  (r, i)
```

Fig. 18. Implementation of a two-way partition in Futhark.

3.2.3 *Implementation of Partition (Filter).*

We have already presented the type and semantics of **filter**. The question is: "can filter be implemented in terms of **map**, **scan**, and **scatter**?" The answer is positive!

In the following we will derive the implementation of a slightly more complicated construct than filter, named **partition**, which has type:

`partition2 : (α → bool) → Π n. [n]α → ([n]α,`**`i32`**`)`

Partition is similar to **filter**, in that it receives as arguments a predicate and an input array, but it returns an array of the same length as the input array and an integer. The array result contains the same elements as the input array, but in a different order:

- the elements that succeed under the predicate come before the ones that fail the predicate,
- the relative order of the elements in the two subarrays is the same as in the original array.

The scalar (integral) result is the number of elements that succeed under the predicate.

The Futhark implementation is shown in fig. 18 and the right hand side of fig. 18 demonstrates the code on an example. The implementation proceeds as follows:

- First, the predicate is mapped on the input array, and the result is turned into ones (for **true**) or zeros (for **false**), which are stored in array `tfs`.
- An inclusive scan with addition is performed on `tfs` resulting in array `isT`. Please observe that array `isT` now holds the value of the indices at which the elements that succeed under the predicate should appear in the result array (plus one). Also the last element of `isT`, saved under variable `i` is the number of elements that succeed under the predicate.
- Array `isF` is computed in a similar fashion and holds the value of the indices at which the elements that fail under the predicate should appear in the result array (plus one).

- The two pieces of information are combined together in array `inds` from arrays `isT` and `isF` (and `cs`) by means of a `map3` operator.
- Finally, the **`scatter`** operator is used to permute the array by `inds`, and the result is written in the new, uninitialized array created by `scratch`.
- The result is the permuted array, tupled with integer `i`, which denotes the number of elements that have succeed under the predicate.

We observe that the implementation is a bit inefficient in that the last **`let`** instruction scatters the elements in a **copy** of the array. The copying requires $2 \cdot n$ memory accesses and is not necessary in an imperative implementation since all n elements are going to be over-written, i.e., the scatter can be safely performed on a freshly allocated array with uninitialized elements. Such a construct, named `scratch` is available in the compiler IR, but is not exposed to the user because Futhark has a deterministic semantics. One can alleviate the overhead by replicating a `dummy` element of type `t` — which would require only n accesses to memory instead of $2 \cdot n$ — but would also require `dummy` to be passed as argument to `partition2` thus polluting a bit its type signature.

3.2.4 *Sparse-Matrix Vector Multiplication.*

Assuming a dense $m \times n$ matrix M and a dense vector v of size n, matrix-vector multiplication can be described by the formula: $r[i] = \Sigma_{j=1}^{n} M[i,j] \times v[j]$, where $i = 0, \ldots m-1$, and r denotes the resulting vector of length m. Dense-matrix vector multiplication can be written in Futhark as:

```
let dnsMatVctMul [m][n] (mat: [m][n]f32) (vct: [n]f32) : [m]f32 =
    map (\row -> let ps = map2 (*) row vct
                 in  reduce (+) 0.0f32 ps   ) mat
```

However, our example refers to *sparse* matrices, whose non-zero values are significantly smaller than the size of the dense matrix; fig. 19 shows several matrix representations of a `m×n` matrix (where `m=5` and `n=4`):

(a) the dense representation is a two-dimensional array of type `[m][n]`**`f32`**. Since the 2-D array is regular—all rows have the same length—the array is assumed to be stored contiguously in a memory space of size `n×m×sizeof(f32)`.

(b) the array-of-pointers sparse representation maintains pointers to the subarrays that represent the rows of the array, except that a row is represented only by the non-zero elements tupled with their corresponding column number. Please note that now the array is irregular, since the number of non-zero element is not the same across all rows.

(c) the flat sparse representation corresponds to a triplet of shape, flag and value arrays. The shape array has size `m` and holds the number of non-zero elements on each row. The flag and value arrays are unidimensional (flat) and their length is equal to the total number of non-zero elements of the matrix. The latter are stored in the value array, while the flag array records a zero at the index of each element that starts a row. This is known as the CSR format.

Futhark supports only regular arrays—i.e., all rows of a matrix have the same length—hence it does not support arrays of pointers. However, assuming a language that supports arrays of pointers (such as Haskell), sparse matrix vector multiplication can be elegantly written using format (b) in the following Futhark-like pseudocode:

```
let spMatVctMul [m][n] (mat:[m][](i32,f32)) (vct:[n]) : [m]f32 =
    map (\row -> let ps = map (\(i,v) -> v * vct[i]) row
                 in  reduce (+) 0.0f32 ps
        ) mat
```

```
-- (a) Dense-Matrix Representation
[ [ 2.0, -1.0,  0.0, 0.0]
, [-1.0,  2.0, -1.0, 0.0]
, [ 0.0, -1.0,  2.0,-1.0]
, [ 0.0,  0.0, -1.0, 2.0]
, [ 0.0,  0.0,  0.0, 3.0]
]

-- (b) Sparse Matrix represented as
-- an array of pointers; each non-zero element
-- records its value and column number:
[ [(0,2.0),  (1,-1.0)],
, [(0,-1.0), (1, 2.0), (2,-1.0)]
, [(1,-1.0), (2, 2.0), (3,-1.0)]
, [(2,-1.0), (3, 2.0)]
, [(3,3.0)]
]

-- (c) Flat Representation of Sparse Matrix:
shape = [2, 3, 3, 2, 1] -- number of non-0 elements of each row
flag  = [1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1]
value = [ (0,2.0), (1,-1.0), (0,-1.0), (1, 2.0), (2,-1.0),
          (1,-1.0), (2,2.0), (3,-1.0), (2,-1.0), (3,2.0), (3,3.0)]
```

Fig. 19. Matrix representations: (a) dense, (b) sparse array of pointers, (c) sparse flat.

Please note that the implementation exhibits irregular parallelism: the size of the inner **map-reduce** operations differs across iterations of the outer **map**. In comparison to the code corresponding to dense matrices, the inner **map** is applied to each row of non-zero elements, whose values are multiplied with their corresponding element `i` in the vector. We will see later, in section 4 how to re-write this irregular nested parallelism by means of (1) a flat-sparse-data representation in format (c), and (2) flat-parallel constructs (that do not exhibit inner parallelism).

3.2.5 *Prime Number Computation (Sieve).*

This section discusses several implementations that, given an integer `n`, compute all the prime natural numbers less than or equal to `n`. This example was introduced in the well-known article “Scan as Primitive Parallel Operation” [Blelloch 1989].

The first implementation, shown in fig. 20, starts with an array `res` of size `n+1`, in which elements at indices 0 and 1 are zero, and the rest of the elements are ones. The meaning is that initially, all natural number greater than 1 are considered prime numbers. Then the implementation iteratively zeros out the array indices corresponding to all multiples of numbers less than or equal to $\sqrt{n}$. This step is accomplished in parallel by means of the **forall** construct. After the sequential loop terminates, the prime numbers are the indices of `res` that hold non-zero (one) values.

```
int res[n+1] = [0, 0, 1, 1, 1, ..., 1]
for(i = 2; i <= sqrt(n); i++)
    if ( res[i] != 0 )
        forall m ∈ multiples of i ≤ n} do
             res[m] = 0;
        endfor
    endif
endfor
```

Fig. 20. Imperative code for the naive computation of prime numbers: Work $O(n\ lg\ lg\ n)$, Suboptimal Depth $O(\sqrt{n})$.

```
-- Primes: Naive Version (primes-naive.fut)
-- ==
-- compiled input { 30i64 }
-- output {[2i64,3i64,5i64,7i64,11i64,13i64,17i64,19i64,23i64,29i64]}
let main (n : i64) : []i64 =                -- Assume n = 9, sq = 3
  let a  = map (\i -> if i==0 || i==1   -- a = [0,0,1,1,1,1,1,1,1,1]
                    then 0 else 1
               ) (iota (n+1))           -- iteration j=0, i=2, m=3
  let sq = i64.f32 (f32.sqrt (f32.i64 n)) -- inds = [4, 6, 8]
  let fl =                                -- vals = [0, 0, 0]
    loop(a) for j < (sq-1) do           -- a'= [0,0,1,1,0,1,0,1,0,1]
      let i    = j + 2
      let m    = (n / i) - 1            -- iteration j=1, i=3, m=2
      let inds = map (\k -> (k+2)*i)    -- inds = [6,9], vals = [0,0]
                     (iota m)           -- a'= [0,0,1,1,0,1,0,1,0,0]
      let vals = replicate m 0
      let a'   = scatter a inds vals    -- iteration j=2, i=4, m=1
      in  a'                            -- a' unchanged
  in  filter (\i -> fl[i] != 0)
             (iota (n+1))               -- Result: [2,3,5,7]
```

Fig. 21. Futhark code for the naive version of primes: Optimal Work $O(n\ lg\ lg\ n)$, Suboptimal Depth: $O(\sqrt{n})$.

This (first) implementation has optimal work $O(n\ lg\ lg\ n)$ but sub-optimal depth: $O(\sqrt{n})$. The latter can be easily observed: the outer loop iterates sequentially to $\sqrt{n}-1$ times. We will call this the naive implementation (due to its sub-optimal depth).

The complete Futhark code for the naive version is shown in fig. 21; it faithfully implements the imperative pseudocode. Furthermore, the right-hand side of the figure demonstrates the case for n=9. Please note that the parallel **forall** loop has be implemented as a composition between **scatter** and **map**. This is fused by the Futhark compiler, so the generated code is as efficient as the imperative one.

The naive version of primes is a good starting point, but we are unhappy with the fact that the depth is simply too high $O(\sqrt{n})$. Luckily, the solution is not terribly complicated: One can reason

```
let main (n : i32) : []i32 =
  let sqrn_primes   = [2]
  let len  = 2
  let (sqrn_primes,_) =
    loop (sqrn_primes, len) while len < n do
      -- this is "len = min n (len*len)"
      let len = if n / len < len then n else len*len

      let composite = -- uses nested parallelism
            map (\p -> let m   = len / p
                       let arr = map (+2) (iota (m-1))
                       in  map (*p) arr
                ) sqrn_primes
      let not_primes = reduce (++) [] composite -- flattens data

      let flat_size  = length not_primes
      let zero_array = replicate flat_size false
      let mostly_ones= map (> 1) (iota (len+1))
      let prime_flags= scatter mostly_ones not_primes zero_array
      let sqrn_primes= filter (\i-> i>1 && i<=n && prime_flags[i])
                              (iota (len+1))
      in  (sqrn_primes, len)
  in sqrn_primes
```

Fig. 22. Futhark-like (illegal) nested-parallel code for primes: Optimal Work $O(n\, lg\, lg\, n)$ and Depth: $O(lg\, lg\, n)$.

that if the primes p between 2 and $\sqrt{n}$ are known (and stored in array `sqrn_primes`), then we could generate all multiples of those primes at once. In the data parallel language NESL, which supports irregular nested parallelism, this computation could be expressed by means of the array comprehension `{[2*p : n : p] : p in sqrn_primes}` which semantically results in an array of arrays, in which each subarray corresponds to one of the known prime numbers between 2 and $\sqrt{n}$. For a given `p`, its subarray consists of the elements `[2*p, 3*p, 4*p, ...]` which are less than or equal to `n`, i.e., the slice that starts from `2*p` and goes with a stride equal to `p`.

A Futhark-like pseudocode is shown in fig. 22—please remember that Futhark does not support irregular arrays/parallelism, so this implementation would not even compile with Futhark; we just use it for consistency. The implementation starts with a known set of primes `[2]` less than or equal to `len=2`. Each iteration of the loop computes a new set of primes less than or equal to `len`^{2}:

- The `composite` array is computed by means of nested-parallelism and it contains all the multiples of the currently known set of primes. The code is semantically equivalent to the previously-discussed NESL array comprehension: `{[2*p : n : p] : p in sqrn_primes}`.
- The `not_primes` array is the flattened version of `composite`, which is an array of arrays.
- The `prime_flags` array is computed by a **`scatter`** operator that
  - writes into an array consisting of mostly one (true) values
  - at the indices corresponding to the computed multiples of prime numbers
  - zero (false) values that indicate that those positions do *not* correspond to prime numbers.
- the **`filter`** operator extracts the indices that correspond to prime numbers (true/one values).

We will not demonstrate how the nested parallel implementation works on a simple example in which `n` is 9. Initially, `len=2` and `sqrn_primes = [2]`. In the first iteration of the loop: `len` is set to $2^2 = 4$, and `composite = [[2*2]] = [[4]]` because there is only one prime `p=2` for which `m=4/2=2` and `arr = [2]`. As such `prime_flags = `**`scatter`**` [0,0,1,1,1] [4] [0] = [0,0,1,1,0]` and the result of the filter is thus `[2,3]`—the set of primes to be used for the next iteration.

The second iteration initially has `len=4` and `sqrn_primes = [2,3]`. Then `len` is set to `9`. Since there are two primes `2` and `3`, the `composite` array is computed as `[[4,6,8],[6,9]]`, where the first subarray corresponds to `p=2` and `m=9/2=4` and the second subarray corresponds to `p=3` and `m=9/3=3`. It follows that `not_primes = [4,6,8,6,9]` and `prime_flags` is computed as
`scatter`` [0,0,1,1,1,1,1,1,1,1] [4,6,8,6,9] [0,0,0,0,0]`
which results in `[0,0,1,1,0,1,0,1,0,0]`, hence the primes less than or equal to 9 are extracted by the **`filter`** operation as: `[2,3,5,7]`.

It remains now to verify that this implementation has improved the depth and by how much. The depth corresponds to the count of the sequential loop—meaning we need to answer the question: "how many iterations does the sequential loop has?" One can observe that `len` starts at $2 = 2^{2^0}$ and each iteration squares up the value of `len`: in the first iteration `len` is 2^{2^1}, in the second is $(2^2)^2 = 2^{2^2}$, in the third is $(2^4)^2 = 2^8 = 2^{2^3}$, hence one can easily prove by induction that in some iteration k the value of `len` is 2^{2^k}. It follows that the total number of iterations of the loop is the first k such that $2^{2^k} \geq n$; it follows that the depth of the new nested-parallel version is $O(lg\ lg\ n)$.

```
let isSorted [n] (arr: [n]i32) : bool =
  reduce (&&) true <|
  map (\i -> i == 0 || arr[i-1] <= arr[i]) (iota n)

let nestedQuicksort [n] (arr: [n]i32) : [n]i32 =
  if n <= 1 || isSorted arr
  then arr
  else let i  = getRand (0, (length arr) - 1)
       -- ^ pseudocode for getting a random integer
       --   (not valid in Futhark)
       let a  = arr[i]
       let s1 = filter (\x -> (x <  a)) arr
       let s2 = filter (\x -> (x >= a)) arr
       let rs = map nestedQuicksort [s1, s2]
       -- ^ recursion not supported in Futhark
       in  (rs[0]) ++ (rs[1])
```

Fig. 23. Futhark-like nested-parallel code for quicksort. (Please be aware that this code is illegal in Futhark!)

3.2.6 *Quicksort.*

A Futhark-like nested-parallel pseudocode for quicksort is shown in Figure 23:

- Please note that this is illegal Futhark code, that will fail compilation because of two reasons: (1) Futhark does not support recursion and (2) the array `[s1,s2]` passed to the recursive call is irregular—the two subarrays do not necessarily have the same length.
- The implementation picks a random pivot `a` and uses the **`filter`** operator to split the array into two subarrays: one containing the elements less than the pivot and one containing the other elements.
- The two subarrays are recursively processed (sorted) by the recursive call `map nestedQuicksort [s1, s2]` and the sorted results are concatenated together to form the sorted array. (The partitioning has already ensured that all the elements of the first subarray are necessarily smaller than the ones of the second.) Please note that the call `map nestedQuicksort [s1, s2]` gives raise to nested parallelism: the divide-and-conquer nature gives raise to a tree in which, in principle, **`filter`** operations can be applied in parallel on all the nodes at the same breadth level in the tree.
- The recursion terminates when the length of the list is less than or equal to 1—a one-element list is always sorted—or when the array is already sorted. The latter is checked with the `isSorted` function, which is implemented by means of a **`map-reduce`** composition.
- Please note that the use of `isSorted` is not an optimization; it is actually necessary to ensure termination. A typical quicksort implementation would perform a three-way partitioning of the array: the elements less than, equal to and greater than the pivot. For simplicity, the presented implementation uses a two-way partition, but this may end up with an array of length > 1 containing the same element, which makes the sorted condition necessary.
- Finally, the three-way splitting version has average work complexity $n\ lg\ n$ and average depth $lg\ n$. The later assumes that **`filter`** has depth $O(1)$; in practice the average depth complexity is $lg^2\ n$.

We conclude by demonstrating quicksort's execution on the simple example when the input array is `arr = [3,2,4,1]`. Assume random `i = 0`, hence `a = 3`. It follows that the array is partitioned into two subarrays, one `s1 = [2,1]` which has elements less than `3`, and the other `s2 = [3,4]` which has its elements greater or equal to three. Next quicksort is (mapped) performed on the two subarrays.

In the case of `nestedQuicksort [2,1]`, assume we pick `i=0`, leading to `a = 2` and we partition `[2,1]` into subarrays `[1]` and `[2]`. These are recursively processed but they hit the base-case since they have length equal to 1, hence they are returned without modification and concatenated into sorted array `[1,2]`.

The case of `nestedQuicksort [3,4]` hits the base case, since it succeeds under the `isSorted` predicate.

The final step is to concatenate the results of the two calls to `nestedQuicksort`, resulting in sorted array `[1,2] ++ [3,4] = [1,2,3,4]`!

4 THE FLATTENING TRANSFORMATION

We have seen in the previous section how non-trivial applications can be naturally constructed by combining parallel operators at the same level or at different levels in a parallel nest. We have also seen that nested parallelism allows to reason asymptotically about the parallel behavior of the implementation, such as its work and depth.

However exploiting nested parallelism is notoriously difficult. Direct utilization of nested parallelism may be possible on some hardware, such as CPU. For example the parallelism of quicksort can be exploited by *dynamically* spawning threads at each divide and conquer step. This technique however is not guaranteed to result in good performance, for example because the distribution of work across threads may be very unbalanced.

More important, a big class of highly-parallel hardware, such as GPUs support only very limited forms of recursion and dynamic parallelism, if at all! Morally, the hardware execution is organized on one (or maybe two) flat-parallel levels—for example, on GPUs one typically exploits grid-level parallelism, and occasionally block-level parallelism, which allows threads within a block-group to communicate by means of shared (scratchpad) memory. This means that direct mapping of application parallelism will require a choice of which level to parallelize and which to sequentialize, because it is not directly-possible to parallelize both levels.

It follows that there is a big disconnect between the nested-parallel form of the program—which has the advantage that it resembles well the algorithmic specification—and a semantically equivalent form of the program that can be efficiently and statically mapped (executed) on highly-parallel hardware. The latter form might be efficient to execute, but likely it resembles little the original algorithm and causes modularity and maintainability issues. Ideally, the re-writing should be done automatically by the compiler, thus getting the best of the two worlds.

This section presents the intuition behind the seminal work on the NESL data-parallel language [Blelloch 1996] related to the flattening transformation [Blelloch et al. 1994] that

- *statically* transforms an arbitrarily-nested data-parallel program into a semantically equivalent one that uses only flat-parallel construct (no nesting of parallelism),
- in a way that preserves the work and depth asymptotic of the original nested-parallel program.

The flattening transformation does not (completely) solve the problem, for example because it requires high memory usage and does not account for communication costs; in fact it often prevents opportunities for locality optimizations, because of excessive utilization of parallelism in excess of what the hardware can support.[9]

However, what flattening primarily offers is a systematic way of reasoning about and transforming nested parallelism. The goal of this chapter is not necessarily to formally introduce the flattening transformation, but instead ***to acquire the touch-and-feel of how it works***. The intent is to train you by means of practical examples, so that you will acquire a sufficient understanding that would allow you to apply its principles in future practical work related to GPU parallelization (or other highly-parallel hardware).

This section is organized as follows:

- section 4.1 presents an ***incomplete set of rules*** related to the flattening of specific code patterns.

[9]Various efforts have focused on addressing these issues by restricting flattening in various ways, for example by flattening only the data and leaving the nested-parallel structure intact [Bergstrom et al. 2013], by applying flattening at the granularity of the largest sequential subexpression [Keller et al. 2012], by aggressive fusion of segmented operations enabled by shape analysis [Reppy and Sandler 2015], or by mechanisms for streaming irregular arrays [Clifton-Everest et al. 2017; Madsen and Filinski 2016] that optimize memory footprint. NESL has also been implemented on GPU hardware [Bergstrom and Reppy 2012].

```
let mkFlagArray 't [m]
            (aoa_shp: [m]i64) (zero: t)   --aoa_shp=[0,3,1,0,4,2,0]
            (aoa_val: [m]t  ) : []t   =   --aoa_val=[1,1,1,1,1,1,1]
  let shp_rot = map (\i->if i==0 then 0   --shp_rot=[0,0,3,1,0,4,2]
                         else aoa_shp[i-1]
                    ) (iota m)
  let shp_scn = scan (+) 0 shp_rot       --shp_scn=[0,0,3,4,4,8,10]
  let aoa_len = shp_scn[m-1]+aoa_shp[m-1]--aoa_len= 10
  let shp_ind = map2 (\shp ind ->        --shp_ind=
                       if shp==0 then -1 --  [-1,0,3,-1,4,8,-1]
                       else ind          --scatter
                    ) aoa_shp shp_scn    --   [0,0,0,0,0,0,0,0,0,0]
  in scatter (replicate aoa_len zero)    --   [-1,0,3,-1,4,8,-1]
             shp_ind aoa_val             --   [1,1,1,1,1,1,1]
                                     -- res = [1,0,0,1,1,0,0,0,1,0]
```

Fig. 24. Constructing the flag array from the shape array

- section 4.2 demonstrates how the previously-discussed rules can be combined to flatten a trivial program.
- section 4.3 advises on how to start reasoning about flattening the nested-parallel versions of sparse-matrix vector multiplication, prime-number computation, and quicksort. These applications were introduced in sections 3.2.4, 3.2.5 and 3.2.6, respectively. The first two are weekly-assignment tasks, and the latter may be a group project.

4.1 Rules For Flattening

We present first the intuition behind the flat-data representation (section 4.1.1), then we present the rules for flattening several constructs which are directly nested inside a map: scan, map, replicate, iota, if-then-else expressions, and reduce.

4.1.1 Data Flattening (Flat Array Representation).

A two-dimensional irregular array—think array of pointers or list of lists—can be represented in a flat way by means of a shape array and a data array, which are both unidimensional arrays that have contiguous support in memory and hence are suitable for data-parallel programming.

For example, the list of lists: `aoa = [[`a^1_1`,...,` $a^1_{m_1}$`],...,[`a^r_1`,...,` $a^r_{m_r}$`]]` can be represented

(1) by the shape array `aoa_shp = [`m_1`,...,` m_r`]`, which specifies the length of each sublist, and
(2) by the flat-data array `aoa_val = [`a^1_1`,...,` $a^1_{m_1}$`, ...,` a^r_1`,...,` $a^r_{m_r}$`]`.

However, we have seen that a segmented scan operator requires a flag array: semantically an array of ones and zeros (booleans), which has the same size as the data array, and which records with a one (true) the element that starts a (new) subarray and zero (false) otherwise. The question is “how do we construct the flag array from the shape array”? fig. 24 provides the implementation together with a side example:

(a) an exclusive scan is performed on the shape array, which is implemented by rotating the array, then by performing an inclusive scan—the result is recorded in `shp_scn`;
(b) the length of the flag array is computed (we assume non-empty shape arrays);

(c) a map is performed on `shp_scn` to compute the indices in the flag array where the one (true) values are to be written—note that if the shape element is `0`, the element is ignored (**`scatter`** ignores `-1` indices);
(d) a **`scatter`** writes into an array of zeroes, at the positions recorded in `shp_ind` the values hold by the third function argument (`aoa_val`); this creates the flag array, denoted by `aoa_flg`.
(e) Instead of simply using an array of ones, the function argument `aoa_val` holds the to-be-written start-of-the-segment values because:
   – any value different than zero may legally denote the start of the segment;
   – sometimes it is useful to have the start of the segment denoted by some value other than 1, for example by the actual length of the corresponding segment. For example if `aoa_val` is the same as `aoa_shp`, this would result in flag array: `[3,0,0,1,4,0,0,0,2,0]`.

Next, we will play with distributing various information to each member of the data array:

(f) "How can we record for each data member the size of its corresponding subarray?"
With our example, the result we seek would be `[3,3,3,1,4,4,4,4,2,2]`. This can be simply accomplished by a segmented inclusive scan with addition on the format (e) of the flag array:

```
let aoa_flag = mkFlagArray aoa_shp 0 aoa_shp
in  sgmScan (+) 0 aoa_flg aoa_flg
```

(g) "How can we record for each data member the index of its corresponding subarray?"
With our example the result we seek would be `[1,1,1,2,4,4,4,4,5,5]`. This can be accomplished by using the values of `iota` plus one for the `aoa_val` parameter, and then by a segmented scan:

```
let iotap1   = map (+1) (iota m)
let aoa_flag = mkFlagArray aoa_shp 0 iotap1
let aoa_iot  = map (-1) aoa_flag
in  sgmScan (+) 0 aoa_flg aoa_iot
```

In the code above, `iotap1 = [1,2,3,4,5,6,7,8]`—it is necessary to add one, otherwise the first start array would be potentially marked by a zero—`aoa_floag = [2,0,0,3,5,0,0,0,6,0]`, hence `aoa_iot = [1,0,0,2,4,0,0,0,5,0]` and finally the segmented scan spreads out the start-of-segment value to the rest of elements in the segment, resulting in the desired `[1,1,1,2,4,4,4,4,5,5]`.

While we will mostly work with two-dimensional irregular arrays, we conclude this section by reasoning about how to generalize the flat representation for a k-dimensional array. This can be simply achieved by recording $k - 1$ shape arrays, i.e., one for each outer dimension. For example, the three-dimensional array:

```
[ [[1,2,3], [4,5], [6,7]], [[9], [8,7], [6], [5,4,3,2]] ]
```

is represented by the shape arrays `aoa_shp0` and `aoa_shp1` and by the flat-data array `aoa_val`:

```
aoa_shp0 = [3, 4]
aoa_shp1 = [3,2,2,1,2,1,4]
aoa_val  = [1,2,3,4,5,6,7,9,8,7,6,5,4,3,2]
```

If required, we can create flag arrays for each dimension, for example from `aoa_shp0` one can compute `aoa_flg0 = [1,0,0,1,0,0,0]` and from `aoa_shp1` one can compute `aoa_flg1 = [1,0,0,1,0,1,0,1,1,0,1,1,0,0,0]`, by using the `mkFlagArray` function shown in fig. 24. Furthermore, one can distribute various information across the data elements of the arrays in a similar fashion with the one used for the two-dimensional arrays.

Exercise 1 (Flattening the inner dimensions of a 3-D array).
Assume a three-dimensional array. Write a function that flattens out the two-inner dimensions of the array. Note that the data array remains the same; what needs to be done is to compute the shape of the resulting array. Assume that the shape and flags for every dimension are available (as arguments).

```
let flatten3to2d [n][m] (aoa_shp0: [m]i32)
                        (aoa_flg0: [n]i32)
                        (aoa_shp1: [n]i32) : [m]i32 = ...
```

In the example just above, the result array should correspond to the list of lists `[ [1,2,3,4,5,6,7], [9,8,7,6,5,4,3,2] ]`, *whose shape should be* `[7,8]`.

In the following subsections related to flattening we will use the following notation: for some two-dimensional irregular array `array_name`, then the shape, flag, and data arrays of its flat representation will be named `array_name_shp`, `array_name_flg` and `array_name_val`, respectively. We will assume that the latter arrays are already available, since this section has shown how they can be computed. We also assume for simplicity that the shape arrays do not contain zero elements. The flattening transformation will be denoted by symbol $\mathcal{F}$.

4.1.2 *Flattening a Scan Directly Nested in a Map.*

RULE 1 ($\mathcal{F}$(MAP(SCAN))).
Flattening a scan that is directly nested inside a map is translated to a segmented scan. This corresponds to computing the data array of the result; the shape and the flag arrays are the same as those of the input array.

```
𝓕( map (\row -> scan (⊙) e⊙ row) A ) ⇒
   ( A_shp, sgmScan (⊙) e⊙ A_flg A_val )
```

We demonstrate this rule for the case when ⊙ = +, and A = [[1,3], [2,4,6]]. The computation of the nested parallel program is:

```
map (\row -> scan (+) 0 row) [[1,3], [2,4,6]] ≡
[ scan (+) 0 [1,3], scan (+) 0 [2,4,6] ] ≡
[ [1, 4], [2, 6, 12] ]
```

The flat representation of A is A_shp = [2,3], A_flg=[1,0,1,0,0], A_val = [1,3,2,4,6]. The computation of the translated, flat parallel program is:

```
sgmScan (⊙) e⊙ [1, 0, 1, 0, 0]
               [1, 3, 2, 4, 6] ≡
               [1, 4, 2, 6, 12]
```

It is trivial to see that a segmented scan preserves the shape: the result array should have the same shape as the input, and hence the same flags.

4.1.3 *Flattening a Map Directly Nested in a Map.*

RULE 2 ($\mathcal{F}$(MAP(MAP))).
Flattening a map that is directly nested inside a map is translated to a map on the flattened data. This corresponds to the data array of the result; the shape and the flag arrays are the same as those of the input array.

```
𝓕(map (\row -> map f row) A) ⇒ (A_shp, map f A_val)
```

We demonstrate this rule for the case when `A = [[1,3], [2,4,6]]`. The computation of the nested parallel program is:

```
map (\row -> map f row) [[1,3], [2,4,6]] ≡
[ map f [1,3], map f [2,4,6] ] ≡
[ [f 1, f 3], [f 2, f 4, f 6] ]
```

The flat representation of A is `A_shp = [2,3]`, `A_flg=[1,0,1,0,0]`, `A_val = [1,3,2,4,6]`. The computation of the translated, flat parallel program is:

```
map f [1,3,2,4,6] ≡ [f 1, f 3, f 2, f 4, f 6]
```

It is trivial to see that the result array should have the same shape, and hence the same flags, as the input array.

4.1.4 *Flattening a Replicate Directly Nested in a Map.*

RULE 3 ($\mathcal{F}$(MAP(REPLICATE))).
Assuming that the values to be replicated are numeric, a replicate that is directly nested inside a map is translated to the following code:

```
𝓕map2 (\ n v -> replicate n v) ns vs ⇒
  ( ns,
    let (flag_n,flag_v) = zip ns vs |> mkFlagArray ns (0,0) |>unzip
    in  sgmScan^inc (+) 0 flag_n flag_v
  )
```

The shape of the result is the array ns, from which one can compute the flag array if needed later. The data array is computed by:

- calling mkFlagArray on its shape ns and the values to be inscribed at start-of-the-index position being an array of tuples: one from ns and one from vs. This results in two "flag" arrays having at the start-of-segment position the elements of ns and vs, respectively. Please note that only the first one can be truely used as a flag array, because vs may contain the value zero, hence it would denote the start of a segment with zero, which would be incorrect;
- distributing the start-of-the-segment value of each segment in flag_v to all remaining elements of the segment. This is accomplished with the inclusive segmented scan with addition operator in which the flag array is flag_n and the value array is flag_v. Note that the implementation assumes the version of inclusive segmented scan that uses a integral representation of the flag array, in which a number different than zero denote the start of a segment.

We demonstrate the algorithm on the simple instance in which ns=[1,3,2] and vs = [7,8,9]. The computation of the nested parallel code is:

```
map2 (\ n v -> replicate n v) [1, 3, 2] [7,8,9] ≡
[ replicate 1 7, replicate 3 8, replicate 2 9 ] ≡
[ [7], [8, 8, 8], [9, 9] ]
```

The translation says that the shape of the result is equal to ns = [1,3,2], which is certainly the case. The computation of the data array by the flat parallel program is demonstrated below:

```
let len  = length ns                    -- 3
let (flag_n, flag_v) =                  -- flag_n = [1,3,0,0,2,0]
                  unzip <|              -- flag_v = [7,8,0,0,9,0]
                  mkFlagArray ns (0,0) <|
                  zip ns vs
in  sgmScan^inc (+) 0 flag_n flag_v -- [7,8,8,8,9,9]
```

It remains to discuss what happens when the element type is not numeric: most of the translated code is the same, except for the zero in the second argument of mkFlagArray, and the associative operator and neutral element of **sgmScan**. If the element type is a:

bool: then we can use false instead of 0 and logical or as operator;
tuple: of numeric types, then we suitably modify the **sgmScan** operator—e.g., for a tuple of integers: ((x1,y1) (x2,y2) -> (x1+x2, y1+y2)).
array: then the inner replicate can be re-written in terms of map and iota as:

```
    replicate n v ≡ map (\i -> v[i % (length v)])
                        (iota (n * (length v)))
```

and flattening will be applied to the rewritten code.

If the element type is some strange scalar type that cannot be translated to numeric, there is still a solution, albeit ugly: zero can be replaced with any (`dummy`) value of the type, and the plus operator of inclusive segmented scan can be replaced with the `first` binary operator that simply returns the first argument `first (a: α) (b: α) : α = a`. This exploits the fact that the inclusive (segmented) scan does not actually needs a neutral element (but exclusive scan does!) One can check that `first` is associative `first a (first b c) = a = first (first a b) c`.

4.1.5 *Flattening an Iota Directly Nested in a Map.*

The main intuition is that `iota n` can be expressed by a composition of scan exclusive and replicate, hence a translation can be derived from the flattening rules of **`scan`** and **`replicate`**:

```
iota n ≡ scan^exc (+) 0 (replicate n 1)
```

However, a much simpler translation can be achieved by high level reasoning, so it is worth to write a specialized rule for **`replicate`**.

RULE 4 ($\mathcal{F}$(MAP(IOTA))).
Flattening a iota that is directly nested inside a map results in an array of the same shape as the argument of map, and in a data array which is computed by the following flat-parallel code:

```
𝓕( map (\n -> iota n) ns ) ⇒
  ( ns,
    let len  = length ns
    let flag = mkFlagArray ns 0 ns
    let vals = map (\ f -> if f!=0 then 0 else 1) flag
    in  sgmScan^inc (+) 0 flag vals
  )
```

We demonstrate the rule for **`iota`** by a simple example in which `ns = [1,3,2]`.

```
map (\ n -> iota n) [1,3,2] ≡
[ iota 1, iota 3, iota 2 ] ≡
[ [0], [0, 1, 2], [0, 1] ]
```

The flat parallel program results in an array whose shape is equal to `ns = [1,3,2]`, which is how it should be. The flat-parallel computation of the value array requires:

- creating the flag array, then
- creating an array in which the value corresponding to the start of the segment is `0`, and 1 otherwise, and
- performing a segmented scan on the two previous arrays.

This is demonstrated below for the case `ns = [1,3,2]`:

```
let len  = length ns                          -- 3
let flag = mkFlagArray ns 0 (replicate len 1) -- [1,3,0,0,2,0]
let vals = map (\f -> ... ) flag              -- [0,0,1,1,0,1]
in  sgmScan^inc (+) 0 flag vals               -- [0,0,1,2,0,1]
```

4.1.6 *Flattening a Reduce Directly Nested in a Map (Segmented Reduce).*

RULE 5 ($\mathcal{F}$(MAP(REDUCE))).
Assume an irregular array of arrays `mat` *of shape* `mat_shp : [num_rows]`**`i64`** *and flat values* `mat_vals :` `[n]`α *(*`n` *is equal to the sum of the elements of* `mat_shp`*) and an associative operator* $\odot : \alpha \rightarrow \alpha \rightarrow \alpha$*, with neutral element* $e_\odot : \alpha$*. An irregular segmented reduce on* `mat` *will be translated to an array whose length is equal to the outer length (i.e., number of rows) of* `mat`*, and whose data is computed by:*

```
𝓕( map (\ row -> reduce ⊙ e⊙ row) mat )⇒
    let mat_flg'= mkFlagArray mat_shp 0 (replicate num_rows true)
    let mat_flg = mat_flg' :> [n]bool  -- dynamic size cast
    let sc_mat  = sgmScan^inc ⊙ e⊙ mat_flg mat_val
    let indsp1  = scan^inc (+) 0 mat_shp
    let res = map2 (\shp ip1 -> if shp==0 then e⊙
                                else sc_mat[ip1-1]
                   ) mat_shp indsp1
    in  ( num_rows, res )
```

In essence, a scan inclusive on the shape of the input array computes the index of the last element in each segment plus one, then a segmented inclusive scan is performed on the data with the operator and neutral element of the reduce and finally, a map operation selects the last element of the segment—this is because the last element of an inclusive scan, by definition, is the reduction of the whole array (segment). Please note that in order to accomodate empty rows, the final **`map`** checks whther the corresponding shape elements is zero, in which case $e_\odot$ is returned—because a **`reduce`** applied to the empty list is by definition supposed to result in the empty element.

Note that in order to get decent performance, the associative-binary operators of reduce/scan/segmented scan should be rewritten whenever possible to operate on scalar types (i.e., a reduce with a vectorized addition would be extremely inefficient).

We demonstrate the translation on a simple example in which our irregular array is `[[1,3,4], [], [6,7]]` and the reduce operator is addition. The nested parallel program computes:

```
map (\ row -> reduce (+) 0 row) [[1,3,4], [], [6,7]] ≡
[ reduce (+) 0 [1,3,4], reduce (+) 0 [], reduce (+) 0 [6,7] ] ≡
[ 8, 0, 13 ]
```

The flat-data representation of the input array is `mat_shp = [3,0,2]`, `mat_val = [1,3,4,6,7]`, `mat_flg = [1,0,0,1,0]`. The flat parallel program results in a uni-dimensional array whose shape is `[3]`. The computation of the data array is:

```
                                                -- mat_shp = [3,0,2]
  let mat_flg = mkFlagArray ...                 -- [1,0,0,1, 0]
  let sc_mat  = sgmScan^inc (+) 0 mat_flg       -- [1,0,0,1, 0]
                                  mat_val       -- [1,3,4,6, 7]
                                                -- [1,4,8,6,13]
  let indsp1= scan^inc (+) 0 mat_shp            -- [3,3,5]
  let res = map2 (\shp ip1 ->                   -- [8,0,13]
                    if shp == 0 then 0
                    else sc_mat[ip1-1]
                  ) mat_shp indsp1
```

4.1.7 *Flattening an If-Then-Else Directly Nested in a Map.*

The rule assumes an **if-then-else** expression, such that the **then** or/and **else** expressions contain parallel constructs; otherwise flattening it is not profitable (does not enhances the degree of parallelism).

RULE 6 ($\mathcal{F}$(MAP(IF-THEN-ELSE))).
Assuming the input array xs *is a uni-dimensional array, and for simplicity, assuming α and β basic (scalar) types, a specialized flattening rule is given below:*

```
𝓕( map (\(x: α) : β -> if p x then f x else g x) xs ) ⇒
  ( [length xs],
    let len = length xs
    let (is, q) = partition2 (\i -> p (x[i])) (iota len)
    let (is^then, is^else) = split q is
    let xs^then = map (\i^then -> xs[i^then]) is^then
    let res^then= 𝓕 ( map f xs^then )
    let xs^else = map (\i^then -> xs[i^else]) is^else
    let res^else= 𝓕 ( map g xs^else )
    let res = scatter (replicate len dummy) is^then xs^then
    in  scatter res is^else xs^else
  )
```

The data array of the result is computed as follows:

- the iteration space of the input array (**iota** len) is partitioned (see fig. 18 in section 3.2.3) according to the predicate p which represents the condition of the **if**: the indices corresponding to the elements that succeed under the predicate and take the **then** branch, namely $\texttt{is}^{then}$, appear before the ones that take the **else** branch, namely $\texttt{is}^{else}$.
- the elements that take the **then** branch are selected and processed with the f function; note that the corresponding two **map**s can be fused. Also, in order for the if-then-else flattening to fire (to be beneficial), we assume that f contains inner parallelism; it follows that we need to recursively flatten **map** f. Similar thoughts apply to those that take the **else** branch.
- finally, the **scatter** operations place the results back in the order of the original array. This step is significantly simplified by the assumption that the mapped function of the original specification results in elements of a basic type.

We demonstrate on the simple example in which the predicate succeeds on odd numbers, the function on the then branch is `f x = 2*x`, the function on the else branch is `g x = x - 1`, and the input array is `[3, 4, 6, 7]`. The nested-parallel version results in `[f 3, g 4, g 4, f 3] = [6, 3, 5, 14]`. The flat parallel version executes as follows:

```
    let (is, q) = partition2 (\i -> p (x[i])) (iota len)
    let (is^then, is^else) = split q is                  -- ([0,3], [1,2])
    let xs^then = map (\i^then -> xs[i^then]) is^then   -- [3, 7]
    let res^then= map f xs^then                          -- [6, 14]
    let xs^else = map (\i^then -> xs[i^else]) is^else   -- [4, 6]
    let res^else= map g xs^else                          -- [3, 5]
    let res = scatter (scratch len β) is^then xs^then   -- [6, u, u, 14]
    in  scatter res is^else xs^else                      -- [6, 3, 5, 14]
```

4.1.8 Exercise: Flattening an Index Directly Nested in a Map.

EXERCISE 2 ($\mathcal{F}$(MAP(INDEX))).
Assume an irregular two-dimensional array `mat` *(i.e., rows have different lengths), which has been previously translated to a flat representation consisting of the shape* `mat_shp`*, flag* `mat_flg` *and data* `mat_val` *arrays. Please write the rule that translates the following* **`map`** *that selects from each row of the matrix the element corresponding to the index taken from* `inds`*:*

```
𝓕 map2 (\row i -> row[i]) mat inds ⇒
  ( ...
  )
```

4.2 Flattening a Simple, Contrived Program

We have presented so far, rather informally, a subset of the flattening rules. While we hope that they make sense individually, it is still probably not very clear the manner in which they can be combined to flatten a nested-parallel program. We demonstrate them on a very simple code example:

```
map (\ i -> map (+(i+1)) (iota i) ) arr  -- arr = [1, 2, 3, 4]
```

in which the mapped array is an uni-dimensional array of integers of length n. We will demonstrate the execution in the particular case when `arr = [1, 2, 3, 4]`.

We first perform the nested parallel computation:

```
map (\i -> map (+(i+1)) (iota i)) arr  ≡ -- arr = [1, 2, 3, 4]
[
  map (+(1+1)) (iota 1)  ≡  map (+2) [0]       ≡  [0+2]  ≡  [2]
, map (+(2+1)) (iota 2)  ≡  map (+3) [0,1]     ≡  [3,4]
, map (+(3+1)) (iota 3)  ≡  map (+4) [0,1,2]   ≡  [4,5,6]
, map (+(4+1)) (iota 4)  ≡  map (+5) [0,1,2,3] ≡  [5,6,7,8]
]     ≡
[[2], [3,4], [4,5,6], [5,6,7,8]]
```

We now turn to the task of flattening our nested parallel program. The first observation is that the program is not in a suitable form: it has to be normalized (or desugared) to a form that would permit the application of the previously discussed rules—think three-address code (TAC) form.

In particular, whenever we encounter a variable that is variant inside the outer map construct, and it is used inside the functional argument of an inner parallel construct (invariant), we will modify the code so that it expands that variable with an extra array dimension (by **`replicate`**) and pass it directly as an array argument to the inner-parallel construct. Such is the case of the `map (+(i+1))` call, because `i` is variant in the outer **`map`** but invariant in the inner one. We solve this by expanding `i` (by means of **`replicate`**) to an array that is passed directly as parameter to the inner **`map`**. The semantically-equivalent and normalized code is:

```
map (\ i -> let ip1 = i+1 in
            let iot = (iota i) in
            let ip1r= (replicate i ip1) in
            in  map (+) ip1r iot
    ) arr
```

We are ready for flattening, which corresponds to distributing the outer **`map`** over each "let-statement" in the body. While doing so, we will semantically expand the left-hand side of the let statement with an outer array dimension equal to the size of the outer **`map`**. We assume that we also maintain a context Σ (think symbol table) that binds the variable names declared inside the map to their expanded arrays (obtained by map distribution/fission). Initially, `Σ = [i→arr]`, and we will use the notation `Σ(x)` to get the flattened array corresponding to symbol `x` in the source program. For example `Σ(i) = arr`. When we distribute the outer **`map`** across a statement we will extend the arguments of the map with the expanded arrays corresponding to the symbols that are used inside the current statement—those have been necessarily translated previously. Long story short here is how the flattening is distributed across the statements of the outer **`map`** body:

```
𝓕( map (\ i -> map (+(i+1)) (iota i) ) arr )
                          ≡
let ip1s = 𝓕( map  (\i -> i+1) arr )        -- Σ=[i→arr,ip1→ip1s]
let iots = 𝓕( map  (\i -> (iota i)) arr )
                                   -- Σ=[i→arr,ip1→ip1s,iot→iots]
let ip1rs= 𝓕( map2 (\(i,ip1) -> (replicate i ip1)) arr ip1s )
                         -- Σ=[i→arr,ip1→ip1s,iot→iots,ip1r→ip1rs]
in  𝓕( map2 (\ip1r iot -> map (+) ip1r iot) ip1rs iots )
```

The parameters of the distributed **map** are obtained by querying the symbols used inside the corresponding let statement: for example the first statement uses only `i`, and Σ(`i`) = `arr`, thus the map is over one array argument `arr` and the lambda formal argument is `i`. After the expression of the first statement has been translated, a fresh variable name is produced `ip1s` to store the result, and a new association is added to the symbol table `ip1`→`ip1s`, because the let statement in the original program was producing variable named `ip1`. Similar thoughts apply to the second let statement.

The third statement uses two variables: `i` and `ip1`. Querying the context results in the corresponding expanded arrays `arr` and `ip1s`, which are passed as arguments to the **map2**, and similar for the result expression, which uses source-program variables `ip1r` and `iot`.

Now we proceed to apply individual rules for each expression. The first one does not correspond to any rule, because there is no inner parallelism to flatten, hence the translation is the identity:

```
let ip1s = 𝓕( map  (\i -> i+1) arr )
            ≡
let ip1s = map  (\i -> i+1) arr        -- [2, 3, 4, 5]
```

The second expression corresponds to a iota directly nested inside a map, and we can apply rule 4:

```
let iots = 𝓕( map  (\i -> (iota i)) arr )
                          ≡
let len  = length arr                -- len=4, arr= [1,2,3,4]
let flag = mkFlagArray arr 0 arr     -- [1,2,0,3,0,0,4,0,0,0]
let vals = map (\ f -> if f != 0     -- [0,0,1,0,1,1,0,1,1,1]
                       then 0
                       else 1
               ) flag
let iots = sgmScan^inc (+) 0 flag vals -- [0,0,1,0,1,2,0,1,2,3]
```

The third expression corresponds to a replicate directly nested inside a map, and we can apply rule 3. Please note that in the code below the computation of the `flag` array is redundant—perhaps compute the `(flag, flag')` pair in the previous step, so that `flag` is not computed twice.

```
let ip1rs = 𝓕( map2 (\(i,ip1) -> (replicate i ip1)) arr ip1s )
                          ≡
let (flag, flag') = unzip <|     -- flag  = [1,2,0,3,0,0,4,0,0,0]
    mkFlagArray arr (0,0) <|     -- flag' = [2,3,0,4,0,0,5,0,0,0]
    zip arr ip1s
let ip1rs = sgmScan^inc (+) 0 flag flag' -- [2,3,3,4,4,4,5,5,5,5]
```

Finally, the last expression corresponds to a map directly nested inside an outer map, and we can apply rule 2, which basically says that this case translates to a map on the flatten data:

```
in ℱ( map2 (\ip1r iot -> map2 (+) ip1r iot) ip1rs iots
                      ≡
in  map2 (+) ip1rs iots    -- [2,3,3,4,4,4,5,5,5,5]
                           --  + + + + + + + + + +
                           -- [0,0,1,0,1,2,0,1,2,3]
                           --  = = = = = = = = = =
                           -- [2,3,4,4,5,6,5,6,7,8]
```

We remark that the shape of the program result is actually `arr`. This is derived from the iota/replicate nested inside a map rules, which generate an array whose shape is the input array of the map. Finally, the last rule (map nested inside a map) preserves the shape of the input array(s).

4.3 Flattening Exercises

This section provides hints related to how to apply flattening to the following problems: sparse-matrix vector multiplication (section 4.3.1), prime number computation (section 4.3.2), and quicksort (section 4.3.3).

4.3.1 *Exercise: Flattening Sparse-Matrix Vector Multiplication.*

We recall that sparse matrix-vector multiplication was discussed in section 3.2.4, and it has the following nested parallel implementation:

```
let spMatVctMul [m][n] (mat: [m][](i32,f32)) (vct: [n]) : [m]f32 =
    map (\row -> let ps = map (\(i,v) -> v * vct[i]) row
                 in  reduce (+) 0.0f32 ps
        ) mat
```

where `mat` is assumed to be an irregular two dimensional array—in which rows have different lengths (think list of lists)—whose elements are a tuple formed by (i) an integer denoting the column number at which the non-zero element appears in the dense matrix, and (ii) a numeric value corresponding to the non-zero value of the matrix element. For simplicity, we also assume that the matrix does *not* have any empty rows (i.e., formed only by zero elements).

To flatten this code, we apply the same procedure used for our contrived example described in section 4.2, in which we distribute the outer **map** across the statements of its lambda function (body):

```
let pss = 𝓕( map (\row -> map (\(i,v) -> v * vct[i]) row) mat )
in  𝓕 ( map (\ps -> reduce (+) 0.0f32 ps) pss )
```

It follows that we need to apply two flattening rules, corresponding to

1. flattening a **map** directly nested in outer **map** (see section 4.1.3)
2. flattening a **reduce** directly nested in an outer **map** (see section 4.1.6).

4.3.2 *Exercise: Flattening Prime Number Computation (Sieve).*

We recall that a nested-parallel implementation—for computing all prime numbers less than or equal to an input integer n, and having work $O(n\ lg\ lg\ n)$ and depth $O(lg\ lg\ n)$—has been discussed in fig. 22 of section 3.2.5. The nested parallelism refers to the code:

```
...
let composite = -- uses nested parallelism
map (\ p -> let m = len / p
            let arr = map (+2) ( iota (m -1))
            in map (*p) arr
    ) sqrn_primes
let not_primes = reduce (++) [] composite
...
```

Our goals is to apply the flattening algorithm to the above code. We use the same procedure as for our contrived example described in section 4.2, in which the first step is to “normalize” the code, by

- computing `m-1` and `iota (m-1)` in separate **let** statements,
- normalizing the inner **map** such that `p` is expanded to an array and passed as argument to the inner **map**.

This results in the code below:

```
let composite = -- uses nested parallelism
map (\ p -> let m   = len / p          -- distribute map
            let mm1 = m - 1            -- distribute map
            let iot = iota mm1         -- 𝓕(Map(Iota))
            let arr = map (+2) iot     -- 𝓕(Map(Map))
            let ps  = replicate mm1 p -- 𝓕(Map(Replicate))
            in map2 (*) ps arr         -- 𝓕(Map(Map))
    ) sqrn_primes
let not_primes = reduce (++) [] composite-- noop because composite
                                         -- is in flat-data form
```

What remains is to distribute the outer **map** across the statements of the lambda body, as indicated on the right-hand side of the code:

- $\mathcal{F}$(Map(Map)) refers to the rule for translating a directly-nested inner **map**, as presented in section 4.1.3;
- $\mathcal{F}$(Map(Iota)) refers to the rule for translating a directly-nested inner **iota**, as presented in section 4.1.5;
- $\mathcal{F}$(Map(Replicate)) refers to the rule for translating a directly-nested inner **replicate**, as presented in section 4.1.4.

4.3.3 *Exercise: Flattening Quicksort.*

We recall that a nested-parallel implementation for computing quicksort has been discussed in fig. 23 of section 3.2.6. The main problem is that the implementation exhibits a **map** over the recursively defined function `quicksort`, which is actually the function which we are trying to flatten: `map nestedQuicksort [s1 , s2]`. This case has not been covered by our rules; but do not fret, the treatment is not difficult.

Mapping a recursive function corresponds to lifting the function: this means that we need to construct a function which is semantically similar to a **map** over the current function:

- the original parameters have to be expanded with a new outer-array dimension corresponding to a generic number of instances, denoted `p`, of the original function that will be computed at once (in parallel);
- a outer **map** of size `p` has to be distributed over the original body of the function;
- the **map** over the recursive function should be translated to the lifted function, because they have equivalent semantics by construction.

The (still) nested-parallel definition of the lifted function is shown in fig. 25. One can go about flattening it by following the rules explain in the previous chapter (and by deriving other rules). However, there are several observations that may allow to derive in an easier way the flattened program (at the expense of perhaps not respecting the work and depth asymptotic of the original program):

(1) In order to eschew the complicate rule for an **if** directly nested inside a map, please observe that the all segments actually belong to the same array, and a conservative and safe stop condition for recursion is when the value array of `arrs` is completely sorted. In essence, if the value array of `arrs` is already sorted, then we should just return it. This hints that the implementation of `liftedQuicksort` should be a **while** loop, which terminates when the

```
isSorted [n] (arr: [n]i32) : bool =
  reduce (&&) true <|
  map (\i -> i == 0 || arr[i-1] <= arr[i]) (iota n)

liftedQuicksort [p] (arrs: [p][]i32) : []i32 =
  map (\arr ->
        if n <= 1 || isSorted arr then arr
        else
        let q   = length arr
        let qm1= q - 1
        let i   = getRand (0, qm1)
        let a   = arr[i]
        let s1 = filter (\x -> (x <  a)) arr
        let s2 = filter (\x -> (x >= a)) arr
        let rs = map nestedQuicksort [s1, s2]
        in  (rs[0]) ++ (rs[1])
      ) arrs

quicksort [n] (arr: [n]i32) : [n]i32 = liftedQuicksort ([arr])
```

Fig. 25. Futhark-like (Nested-Parallel) Lifted Quicksort function.
(Please be aware that this code is illegal in Futhark!)

value array is sorted. It remains to distribute the body of the map on the statements belonging to the body of the **then** branch.

(2) the computation of q can be obtained from the shape of `arrs`.
(3) `getRand`—which is supposed to return a random number between `0...qm1`—should be lifted as well; I suggest using a random number generator that does not have state, such as Sobol numbers (ask for help if you chose this as project).
(4) the two **`filter`** operations can be rewritten by means of `partition2`, defined in fig. 18.
(5) the translation of the last two statement:
`let rs = map nestedQuicksort [s1, s2] in rs[0]++rs[1]`
would correspond to creating a new shape array that takes into account the split indices resulted from applying `partition2`, which will be used by the next iteration of the encompassing **while** loop; please notice that `rs[0]++rs[1]` is a noop because the value array is flat anyway.

In essence the function can be rewritten to operate on the flat-data representation, according to the template presented in fig. 26, where $\mathcal{F}$ represents the application of the flattening transformation. The tricky parts remaining to implement/flatten are:

- the flattening of the random-number generator (try sobol), which can be inlined and flattened according to the rules,
- the flattening of the array indexing `arr[i]`—this was left as an exercise in exercise 2 of section 4.1.8,
- the flattening of the `partition2` call, which can be inlined and flattened according to the rules; perhaps this is the most challenging part,

```
liftedQuicksort [m][n] (arr_shp: [m]i32) (arr_val: [n]i32)
                                        : ([m]i32, [n]i32) =
  loop (arr_shp, arr_val)
    while not (isSorted arr_val) do
      let qs   = arr_shp
      let qm1s = map (\q -> q-1) qs
      let is   = ℱ( map  (\qm1 -> getRand (0,qm1)) qm1s )
      let as   = ℱ( map  (\i -> arr[i]) arrs ) -- ℱ(Map(Index))
      let (arr_val', ss) =
          ℱ( map2 (\a arr -> partition2 (\x-> x<a) ) as arrs )
      let arr_shp' = ... ??? ... -- use split points in ss
      in  (arr_shp', arr_val')   -- to create a new shape.

quicksort [n] (arr: [n]i32) : [n]i32 = -- initially one subarray of
  let (_, arr') = liftedQuicksort ([n]) ([arr]) in arr' -- size n
```

Fig. 26. Flat-Parallel Template for Quicksort.

- finally the computation of the shape array corresponding to the segmented partition—this can be done in an intuitive way (i.e., not according to rules).

5 LOOP-BASED DATA-DEPENDENCE ANALYSIS AND APPLICATIONS

So far, we have assumed that the user writes a fresh implementation of a *known* algorithm, and we have demonstrated how “parallel thinking” can

- derive a nested-parallel implementation, which is close, in spirit, to the algorithmic specification—e.g., by faithfully reproducing the control structure of the algorithm—and which answers well software engineering concerns such as modularity, code re-use, etc. In essence, the nested-parallel implementation can be maintained direcly by the (algorithm-) domain expert rather than by the computer scientist who sleeps with the hardware-specification and compiler-transformations books under his pillow.
- reason about the asymptotic work-depth behavior of the nested parallel implementation, and
- automatically derive a low-level, flat-parallel implementation that is suitable to being directly (statically) mapped on highly-parallel hardware, and which preserves the work-depth asymptotic of the nested-parallel implementation (from which it has been derived).

One may rightfully raise the question “Why then do we need to learn how to reason about low-level imperative code written in terms of loops and array accesses?” There are at least two important reasons that justify this direction:

1) There is a lot of legacy (already written) sequential (scientific) code written in imperative languages such as C++, Java, Fortran, and either the precise algorithm to which they correspond to (i) may have been forgotten (not documented), or (ii) a fresh implementation is infeasible (e.g., because it costs too much). At some point you may wish (or be asked) to parallelize such code to run efficiently on a certain hardware. This will require
   a) to identify the computational kernels where most of the runtime is spent, and
   b) to optimize them by reasoning at a lower-level of abstraction about which loops in the nest are parallel, and
   c) the manner in which loop nests can be re-written in order to optimize locality of reference, load balancing, thread divergence, etc.
2) The flattening transformation guarantees preservation of the *asymptotic operational* work-depth behavior of the transformed program, but
   a) it may require *impractically-high memory usage*, which may be asymptotically higher than the one corresponding to the sequential algorithm—for example a fully flattened implementation of matrix-matrix multiplication of $n \times n$ matrices requires $O(n^3)$ memory space in comparison with $O(n^2)$ required by the sequential implementation.
   b) *it does not account for locality of reference and inter-thread communication.* In fact full flattening results in flat-parallel code that does not contain any inner recurrences (think no inner sequential loops), and as such, it actually destroys the program structure necessary for performing locality-based optimizations. For example, the fully flattened implementation of matrix matrix multiplication would exploit all available parallelism but its performance behavior, in practice, will be limited by the bandwidth supported by the underlying hardware. In comparison, an implementation that sequentializes the innermost dot-product operation and performs block tiling in scratchpad memory to optimize locality, would exhibit a compute-bound behavior—in which the limiting factor for performance is the throughput of floating point operations per second supported by the hardware (which is typically much higher than the hardware bandwidth).

It follows that in many cases, especially the ones involving regular parallelism—in which the length of the inner parallel construct is the same across the encompassing (outer) **`map`** operation—full flattening yields very pessimistic performance results. While the style of reasoning introduced by the flattening transformation remains important, this chapter augments it with a set of lower-level, loop-based transformations that address the major shortcomings identified above.

The main source of inspiration for the material presented in this section has been the book "Optimizing compilers for modern architectures: a dependence-based approach" [Kennedy and Allen 2001]. The goal of this chapter is to provide the intuition by discussing a "simple" static analysis applied here to multicore machines; more advanced treatment that combines static and dynamic analysis for automatic parallelization is presented elsewhere [Hall et al. 2005; Moon and Hall 1999; Oancea and Rauchwerger 2011, 2012, 2015].

This section is organized as follows:

- section 5.1 introduces various nomenclature, in particular the notion of cross-iteration dependency, and it shows how to summarize dependencies across the iteration space into a succinct representation, named direction vectors, that promotes reasoning about various code transformations;
- section 5.2 presents a simple theorem that allows easy identification of loop-level parallelism;
- section 5.3 presents a simple theorem that gives necessary conditions for the safety of the transformation that interchanges two perfectly-nested loops;
- section 5.4 discusses the legality and the manner in which a loop can be distributed across the statements in its body. The treatment is a bit more general than flattening, which essentially distributes a parallel loop (**map**) across its statements; we will see that in some cases one can also distribute a dependent (sequential) loop across subsets of its statements;
- section 5.5 discusses techniques for eliminating cross-iteration write-after-read and write-after-write dependencies;
- section 5.6 discusses how one can recognize data-parallel operators hidden in sequential, imperative code and the manner in which these can be parallelized;
- section 5.7 introducing a simple transformation, named stripmining, which is always valid, and shows how block and register tiling can be derived as a combination of stripmining, loop interchange and loop distribution.

5.1 Data-Dependence Analysis: Direction Vectors

We start by recalling the so called data hazards, which should be familiar to the reader from the introductory course in computer architecture. These refer to dependent instructions that must receive special treatment when executing in a pipeline—for example by stalling the pipeline or by forwarding values between pipeline stages—because otherwise their execution might be improperly reordered in a way that contradicts the semantics of the program.

```
-- True Dependency    | Anti Dependency    | Output dependency
--        RAW         |        WAR         |        WAW
S1:    X  = ..        | S1:    .. = X      | S1:    X  = ...
S2:    .. = X         | S2:    X  = ..     | S2:    X  = ...
```

The possible data hazards are depicted in the code above between two statements S1 and S2, which reside for simplicity in the same basic block—a straight-line of three-address code, which is always entered by the first statement and is exited after the execution of the last statement.

RAW: refers to the case when a write to a register or memory location is followed, in program order, by a read from the same register or memory location; this is typically referred to as a read-after-write hazard in hardware-architecture nomenclature, and as a *true* dependency in loop-based analysis nomenclature. The word *true* refers to the fact that such a dependency denotes a producer-consumer relation, in which the value produced by S1 is used in S2. The producer-consumer relation is an algorithmic property of the program; such a dependency cannot be eliminated other than by changing the underlying algorithm.

WAR: refers to the case when a read from a register or memory location is followed, in program order, by a write to the same register or memory location; this is referred to as a write-after-read hazard, and equivalently as an *anti* dependency. The problem here is that, if the two statements are reordered—meaning `S2` executes before `S1`, then the value needed by `S1` is no longer available because it has been already overwritten by `S2`.

WAW: refers to the case when a write from a register or memory location is followed, in program order, by another write to the same register or memory location; this is referred to as a write-after-write hazard, and equivalently as an *output* dependency. The problem here is that, if the two statements are reordered—meaning `S2` executes before `S1`, then the final value stored in register or memory location is that of `S1` rather than that of `S2`.

In what parallelism or loop analysis is concerned, we are primarily interested in analyzing the (true, anti and output) *dependencies that occur across different iterations of the loop.* For example such a true dependency would correspond to the case in which an early iterations `i` writes/produces an array element that is subsequently read/consumed in a later iteration `j > i`.

In what parallelization is concerned, the main limiting factor are the true dependencies—which correspond to an algorithmic property—because the anti and output dependencies can be typically eliminated by various techniques, which will be reviewed at a later time.

5.1.1 Loop Notation and Lexicographic Ordering of Iterations in a Loop Nest.

In the following we will write loop nests using the **do**-loop notation, inspired from Fortran:

```
do i = low_bound, high_bound, stride
    ... loop body ...
enddo
```

because Fortran **do** loops have the property that the loop index `i` cannot the modified inside the loop. As such the first iteration uses `i = low_bound`, the second iteration uses `i = low_bound + stride`, the third uses `i = low_bound + 2*stride`, and so on, for as long as `i ≤ high_bound`. When the stride is one, one may use the abbreviation **do** `i = low_bound, high_bound`.

In the following we will also assume that iterations in a loop nest are represented by a vector, in which iterations numbers are written down from the corresponding outermost to the innermost loop in the nest, and are ordered *lexicographically*—i.e., are ordered consistently with the order in which they are executed in the (sequential) program. This means that in the loop nest below:

```
do i = 0, N-1
  do j = 0, M-1
    ... loop-nest body ...
  enddo
enddo
```

iteration $\overline{k}$=`(i=2,j=4)` is smaller than iteration $\overline{l}$=`(i=3,j=3)` (i.e., $\overline{k} < \overline{l}$), because the second iteration of the outer loop is executed before the third iteration of the outer loop, no matter what the iteration numbers are executed for the inner loop (of index `j`). In essence the iteration numbers of inner loops are only used to discriminate the order in the cases in which all the outer-loop iterations are equal, for example $\overline{k}$=`(i=3,j=3)` $<$ $\overline{l}$=`(i=3,j=4)`

5.1.2 Dependency Definition.

The precise definition of a dependency between two statements located inside a loop nest is given below.

Definition 5 (Loop Dependency).
There is a dependency from statement S_1 to statement S_2 in a loop nest if and only if *there exists loop-nest iterations $\vec{k}$, $\vec{l}$ such that $\vec{k} \leq \vec{l}$ and there exists an execution path from statement $S1$ to statement $S2$* such that:

1. *$S1$ accesses some memory location M in iteration $\vec{k}$, and*
2. *$S2$ accesses the same memory location M in iteration $\vec{l}$, and*
3. *one of these accesses is a write.*

In such a case, we say that S_1 is the source *of the dependence, and that $S2$ is the* sink *of the dependence, because S_1 is supposed to execute before S_2 in the sequential program execution.*
Dependencies can be visually depicted by arrows pointing from the source to the sink of the dependence.

The definition basically says that in order for a dependency to exist, there must be two statements that access *the same memory location* and one of the accesses must be a write—because two read instructions to the same memory location do not produce a data hazard on any architecture that we are aware of. The nomenclature denotes the statement that executes first in the program order as *the source* and the other as *the sink* of the dependency. We represent a dependency graphically with an arrow pointing from the source to the sink.

Optimizations aimed at optimizing instruction-level parallelism (ILP)—meaning eliminating as much as possible the stalls from processor's pipeline execution—typically rely on intra-iteration analyses (i.e., $\overline{k} = \overline{l}$). Higher-level optimizations, such as detection of loop parallelism, are mostly concerned with analyzing inter-iteration dependencies (i.e., $\overline{k} \neq \overline{l}$). For example the main aim could be to disprove the existence of inter-iteration dependencies, such that different iterations may be scheduled out of order (in parallel) on different cores, while the body of an iteration is executed sequentially on the same core. As such, intra-iteration dependencies are trivially preserved, hence are not very interesting in such a context.

5.1.3 Aggregating Dependencies by Means of Direction Vectors.

Assume the three loops presented in fig. 27, which will be used as running example to demonstrate data-dependency analysis and related transformations. We make the important observation that the code is not in three-address code (TAC) form: a statement such as `A[j,i] = A[j,i] + 3` would correspond to three TAC or hardware instructions: one that loads from memory `tmp1 = A[j,i]`, followed by one that performs the arithmetic operation `tmp2 = tmp1 + 3`, followed by one that writes to memory `A[j,i] = tmp2`. Analysis should semantically be carried out (reasoned) on TAC form but, for brevity, our analysis (and also compiler analysis) will be carried out at the statement level. A human may start analyzing dependencies:

- by depicting the iteration space in a rectangle in which the x axis and y axis correspond to iteration numbers of the inner loop j and outer loop i, respectively, and
- then by reasoning point-wise about what dependencies may happen between two iterations.

A graphical representation of the dependencies of the three running code examples is shown in fig. 28. They can be intuitively inferred as follows:

- For the loop in fig. 27(a), different loop-nest iterations (i_1, j_1) and (i_2, j_2) necessarily read and write different array elements `A[`j_1, i_1`]` and `A[`j_2, i_2`]`. This is because the assumption was that

```
do i = 0, N-1          | do i = 1, N-1               | do i = 1, N-1
  do j = 0, N-1        |   do j = 1, N-1             |   do j = 0, N-1
S1: A[j,i]=A[j,i]...   | S1: A[j,i]=A[j-1,i-1]...    | S1: A[i,j] =
  enddo                | S2: B[j,i]=B[j-1,i  ]...    |        A[i-1,j+1]...
enddo                  |   enddo                     |   enddo
                       | enddo                       | enddo
--   (a)               |           (b)               |          (c)
```

Fig. 27. Three Simple Running Code Examples that will be used to demonstrate data-dependency analysis and related transformation.

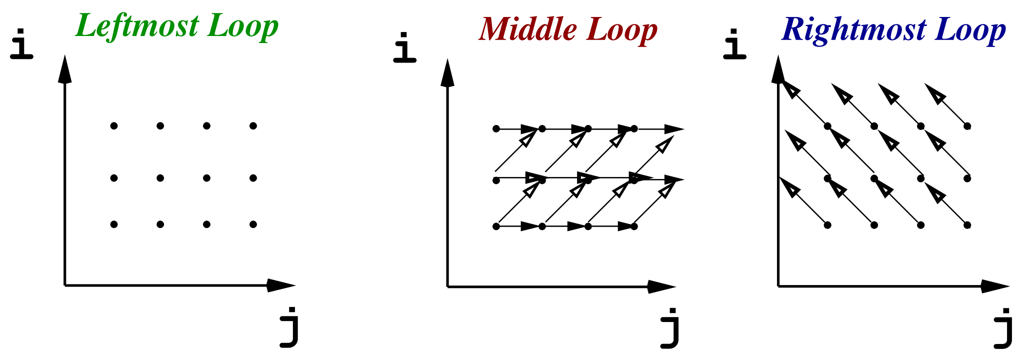


Fig. 28. Graphical representation of the dependencies for the three running examples shown in fig. 27; the x and y axis correspond to the index of the inner and outer **do** loop, respectively.

$(i_1, j_1) \neq (i_2, j_2)$, hence it cannot be that both $i_1 = i_2$ and $j_1 = j_2$. As such, the representation of dependencies should be a set of points (no arrows), meaning that all dependencies actually occur inside the same iteration—in fact they are anti intra-iteration dependencies (WAR) because `A[j,i]` is read first and then `A[j,i]` is written inside the same iteration.

- For the loop in fig. 27(b) we reason individually for statements S_1 and S_2 because each statement accesses (one) array `A` and `B`, respectively:

S_1 : Let's take an iteration, say $(i_1 = 2, j_1 = 3)$, which *reads* element `A[`j_1`-1,`i_1`-1] = A[2,1]`. Since an iteration (i, j) always writes the element `A[i,j]`, we can reason that iteration $(i_2 = 1, j_2 = 2)$ will *write* the same element `A[2,1]`. It follows that we have discovered a true (RAW) dependency, depicted in the figure with and arrow, from the source iteration $(i_2 = 1, j_2 = 2)$—which writes `A[2,1]`—to the sink iteration $(i_1 = 2, j_1 = 3)$—which reads `A[2,1]`. This is because iteration $(1, 2) < (2, 3)$ according to the lexicographical ordering, and as such, the read happens after the write (RAW) in program order. One can individually reason for each point of the iteration space and fill it with oblique, forward-pointing arrows denoting true dependencies between different instances of statement S_1 (executing in different iterations).

S_2 : Following a similar rationale, iteration $(i_1 = 2, j_1 = 3)$ *reads* element `B[`j_1`-1,`i_1`] = B[2,2]`, and iteration $(i_2 = 2, j_2 = 2)$ *writes* element `B[2,2]`. It follows that we have discovered a true (RAW) dependency with source $(i_2 = 2, j_2 = 2)$ and sink $(i_1 = 2, j_1 = 3)$, because $(2, 2) < (2, 3)$ in lexicographic ordering. Since $i_1 = i_2$ we depict the arrow parallel with the horizontal axis (that depicts values of j). One can fill in the rest of the iteration space with horizontal arrows.

- For the loop in fig. 27(c) we reason in a similar way: take iteration ($i_1 = 2, j_1 = 3$) that *reads* element `A[2-1,3+1] = A[1,4]`. This element is *written* by iteration ($i_2 = 1, j_2 = 4$). It follows that we have discovered a true (RAW) from source ($i_2 = 1, j_2 = 4$) to sink ($i_1 = 2, j_1 = 3$)—because the read happens in iteration $(2, 3)$ which comes after the write in iteration $(1, 4)$, i.e., $(1, 4) < (2, 3)$. Thus, one can fill in the iteration space with oblique, backward-pointing arrows, denoting true dependencies between instances of S_1 executing in different iterations.

We have applied above a human type of reasoning and, as a result, we have a graphical representation of all dependencies. However, such a reasoning is not suitable for compiler-based automation because (i) the loop counts are statically unknown—they depend on the dataset—hence one cannot possibly represent an arbitrary large iteration space, and, more importantly, (ii) even if the loop counts would be statically known it is still inefficient to maintain and work with all this pointwise information.

A representation that promotes compiler reasoning should succinctly capture the pattern that is multiplexed in the figure. Intuitively and imprecisely, for fig. 27(a) the pattern would correspond to a point, for fig. 27(b) it would correspond to two arrows—one oblique and one horizontal forward pointing arrows—and for fig. 27(c) it would correspond to an oblique, backward-pointing arrow. These patterns are formalized by introducing the notion of direction vectors, which are defined below.

Definition 6 (Dependency-Direction Vector).
Assume there exists a dependency with source $S1$ in iteration $\overline{k}$ to sink $S2$ in iteration $\overline{l}$ ($\overline{k} \leq \overline{l}$). We denote by `m` *the depth of the loop nest, we use i to range from* $0, \ldots,$`m-1`*, and we denote by x_i the i^{th} element of some vector $\overline{x}$ of length* `m`.

The direction vector *between the instance of statement S_1 executed in some source iteration $\overline{k}$ and statement S_2 executed in sink iteration $\overline{l}$ is denoted by $\overline{D}(S_1 \in \overline{k}, S2 \in \overline{l})$, and corresponds to a vector of length* `m`*, whose elements are defined as:*

$$D_i(S_1 \in \overline{k}, S_2 \in \overline{l}) = \begin{cases} < & \texttt{if it is provably that } k_i < l_i, \\ = & \texttt{if it is provably that } k_i = l_i, \\ > & \texttt{if it is provably that } k_i > l_i, \\ * & \texttt{if } k_i \texttt{ and } l_i \textit{ are statically uncomparable.} \end{cases}$$

The first three cases of the definition above assume that the ordering relation between k_i and l_i can be statically derived in a generic fashion (for any source k_i and l_i); if this is not possible than we use the notation $*$ which *conservatively* assumes that any directions may be possible—i.e., star should be understood as simultaneous existence of all `<, =, >` directions. For example, the loop

```
do i = 0, N-1
$S_1$:  A[ X[i] ] = ...
enddo
```

would result in direction vector `[*]` corresponding to a potential output dependency (WAW), because the write access to `A[ X[i] ]` is statically unanalyzable—for example under the assumption that the indirect array `X` is part of the dataset—and, as such, all direction vectors may possibly hold between various pairs of instances of statement S_1 executed in different iterations.

We also remark that the symbols `<, =, >` are *not* connected at all to the type of the dependency, e.g., true (RAW) or anti (WAR) dependency. The type of the dependency is solely determined by the operation of the source and that of the sink: If the source is a write statement and the sink is a read then we have a true (RAW) dependency; if the source is a read and the sink is a write then we have an anti (WAR) dependency; if both source and sink are writes then we have an output (WAW) dependency.

The meaning of the symbol > at some position i is that the source iteration at loop-level i is greater than the sink iteration at loop-level i. This case is possible, for example the code in fig. 27(c) shows a dependency with source iteration $(1, 4)$ and sink iteration $(2, 3)$. At the level of the second loop, we have $4 > 3$ hence the direction is > but still the source iteration is less than the sink iteration $(1, 4) < (2, 3)$ because of the first loop level. This observation leads to the following corollary:

Corollary 1 (Direction Vector Legality).
A direction vector is legal (well formed), if filtering out the = *entries does* not *result in a leading* > *symbol. This would mean that a current iteration depends on a future iteration, but depending on a future event is (considered) impossible, and as such illegal.*

It remains to determine the sort of automatic reasoning (think compiler reasoning) that can be applied to compute the direction vectors for the code examples in fig. 27:

fig. 27(a): dependencies can occur only between instances of statement S_1, executed in different (or the same) iterations. We recall that, by the definition of dependency, the two (dependent) iterations must access the same element of `A` and at least one iteration should perform a write. Since statement S_1 performs a read and a write to elements of array `A`, two kinds of dependencies may occur:

WAW: an output dependency may be caused by two write accesses in two different iterations, denoted (i_1, j_1) and (i_2, j_2). The written element is thus `A[`j_1`,`i_1`]`, which must be the same as `A[`j_2`,`i_2`]` for a dependency to exist. This results in the system of equations $\begin{cases} i_1 = i_2 \\ j_1 = j_2 \end{cases}$ which leads to direction vector `[=,=]`. Hence, an output dependency from S_1 to S_1 happens in the same iteration, but statement S_1 executes only one write access in the same iteration. The conclusion is that no output dependency can occur, hence the direction vector is discarded.

RAW: a true or anti dependency—we do not know yet which—will be caused by the read access from `A` and the write access to `A` in different (or same) iterations. Remember that a statement such as `A[j,i] = A[j,i] + 3` actually corresponds to three hardware instructions, hence either an inter- or an intra-dependency will necessarily occur. Assume some iteration (i_1, j_1) reads from `A[`j_1`,`i_1`]` and iteration (i_2, j_2) writes to `A[`j_2`,`i_2`]`. In order for a dependency to exist, the memory location of the read and write must coincide; this results in the system of equations: $\begin{cases} i_1 = i_2 \\ j_1 = j_2 \end{cases}$ from which we can derive the direction vector: `[=,=]`. This implies that the dependency happens in the same iteration, hence it is an intra-dependency. Furthermore, since the write follows the read in the instruction order of an iteration, this is an anti dependency (WAR).

fig. 27(b): dependencies may possibly occur between instances of statement S_1 and between instances of statement S_2. The case of output dependencies is disproved by a treatment similar to the bullet above. It remains to examine the dependency caused by a read and a write in different instances of S_1 and S_2, respectively:

S_1: assume iteration (i_1, j_1) and iteration (i_2, j_2) reads from and writes to the same element of `A`, respectively. Putting this in equation results in the system: $\begin{cases} i_1 - 1 = i_2 \\ j_1 - 1 = j_2 \end{cases}$, which necessarily means that $i_1 > i_2$ and $j_1 > j_2$. However, we do not know yet which iteration is the source and which is the sink. Assuming that (i_1, j_1) is the source results in the direction vector `[>,>]`, which is illegal by corollary 1, because a direction vector cannot start with the > symbol. It follows that our assumption was wrong: (i_2, j_2) is the source and (i_1, j_1) is the sink, which means that this is a cross-iteration (inter-iteration) true dependency (RAW)—because the sink iteration reads the element that was previously written by the source iteration—and its direction vector is `[<,<]`.

S_2: a similar rationale can be applied to determine that two instances of S_2 generate a true cross-iteration dependency (RAW), whose direction vector is `[=,<]`. In short, using the same notation results in the system of equations $\begin{cases} i_1 = i_2 \\ j_1 - 1 = j_2 \end{cases}$, hence the source must be (i_2, j_2) and the sink must be (i_1, j_1) and the direction vector is `[=,<]`.

fig. 27(c): dependencies may possibly occur between instances of statement S_1. Assume iteration (i_1, j_1) and (i_2, j_2) reads from and writes to the same element of `A`, respectively. Putting this in equation results in the system $\begin{cases} i_1 - 1 = i_2 \\ j_1 + 1 = j_2 \end{cases}$, which necessarily imply that $i_1 > i_2$ and $j_1 < j_2$. Choosing (i_1, j_1) as the source of the dependency results in direction vector `[>,<]`, which is illegal because it has > as the first non-= outermost symbol, as stated by corollary 1. It follows that (i_1, j_1) must be the sink and (i_2, j_2) must be the source, which results in the direction vector `[<,>]`, which is legal. Since the source writes and the sink reads, then we deal with a true dependency (RAW). Moreover since the direction vector indicates that the source iteration is strictly less than the sink iteration, this is also a cross-iteration dependency.

Definition 7 (Dependency-Direction Matrix). *A direction matrix is obtained by stacking together the direction vectors of all the intra- and cross-iteration dependencies of a loop nest (i.e., between any possible pair of write-write or read-read instruction instances).*

In conclusion the direction matrices for the three running code examples in fig. 27 are:

(a): $\begin{cases} \texttt{[=,=]} \end{cases}$

(b): $\begin{cases} \texttt{[<,<]} \\ \texttt{[=,<]} \end{cases}$

(c): $\begin{cases} \texttt{[<,>]} \end{cases}$

The following section will show how the legality of powerful code transformations can be reasoned in a simple way in terms of direction vectors/matrices.

5.2 Determining Loop Parallelism by Analyzing Direction Vectors

A loop is said to be parallel if its execution does not cause any (true, anti or output) dependencies across its iterations—the loop execution is assumed to be fixed in a specific iteration of an (potentially empty) enclosing loop context.

The following theorem states that a sufficient condition for a loop to be parallel is that for all the elements in the loop's corresponding direction-matrix column, it holds that the element is either = or there exists an outer loop whose corresponding direction is < (on that row). In the latter case we say that the outer loop carries all the dependencies of the inner loop, i.e., fixing an iteration of the outer loop (think executing the outer loop sequentially) would guarantee the absence of cross-iteration dependencies in the inner loop.

Theorem 6 (Parallel Loop).
We assume a loop nest denoted by $\overline{L}$, whose direction matrix is denoted by M and consists of m rows. A sufficient condition *for a loop at depth k in $\overline{L}$, denoted L_k, to be parallel is that $\forall i \in \{0, \ldots m-1\}$ either $M[i,k]$ is equal to = or there exists an outer loop (at depth $q < k$) such that $M[i,q]$ is equal to <.* The proof *is left as an exercise.*

Theorem 6 claims to give only a sufficient condition for loop parallelism because it assumes that symbols such as `*` may be part of the direction vector elements—we recall that `*` conservatively assumes that all directions `<`,`=`,`>` may be possible. If `*` does not appear in the direction matrix, then the condition becomes necessary and sufficient, i.e., the loop is parallel if and only if . . .. Let us analyze the parallelism of each loop in our running examples:

fig. 27(a): The direction matrix is `[=,=]`, hence by theorem 6, both loops in the nest are parallel because all the directions are equal to =.

fig. 27(b): The direction matrix is $M = \begin{cases} [<,<] \\ [=,<] \end{cases}$, hence neither the outer nor the inner loop can be proven parallel by theorem 6. In the former case this is because $M[0,0]$ is equal to < and there is no other outer loop to carry dependencies. In the latter case this is because $M[1,1]$ is equal to < and the outer loop for that row has direction = (instead of <, which would have been necessary to carry the dependencies of the inner loop).

fig. 27(c): The direction matrix is `[<,>]`, which means that the outer loop is not parallel—because it has a leading < direction)—but the *inner loop is parallel* because the outer loop starts with < on the only row of the direction matrix, and, as such, it carries all the dependencies of the inner loop. To understand what this means, take a look again at the actual code in fig. 27(c): let us fix the outer iteration number to some value `i`. Then the read accesses always refer to row `i-1` of matrix `A` and the write accesses always refer to row `i` of `A`; hence a cross-iteration dependency cannot happen in the inner loop because no matter of the value of `j`, the read and write statement instances cannot possibly refer to the same location of `A`.

5.3 Loop Interchange: Legality and Applications

Direction vectors are not used only for proving the parallel nature of loops, but they can also enable powerful code restructuring techniques. For example they can be straightforwardly applied to determine whether it is safe to interchange two loops in a perfect loop nest[10]—which may result in better locality and even in changing an inner loop nature from dependent (sequential) to parallel.

The following theorem gives a sufficient condition for the legality of loop interchange—i.e., for the transformation to result in code that is semantically equivalent to the original one.

Theorem 7 (Legality of Loop Interchange).
A sufficient condition for the legality of interchanging two loops at depth levels k and l in a perfect nest is that interchanging columns k and l in the direction matrix of the loop nest does not result *in a (leading) > direction as the leftmost non-= direction of any row.*

The theorem above shows that the legality of loop interchange can be determined solely by inspecting the result of permuting the direction matrix in the same way as the one desired for loops. For the rationale related to why a row-leading > direction is illegal, we refer the reader to corollary 1: a non-= leading > direction would correspond to depending on something that happens in the future: this currently seems impossible in our universe, and as such it signals an illegal transformation. The following corollary can be easily derived from theorem 7:

[10] A perfect loop nest is a nest in which any two loops at consecutive depth levels are not separated by any other statements; for example all loop nests in fig. 27 are perfectly nested.

Corollary 2 (Interchanging a Parallel Loop Inwards).
In a perfect loop nest, it is always safe to interchange a parallel loop *inwards one step at a time (i.e., if the parallel loop is the k^{th} loop in the nest then one can always interchange it with loop $k + 1$, then with loop $k + 2$, etc.).*

The corollary says that if we somehow know the parallel nature of a loop, then we can safely interchange it in the immediate inward position, without even having to build the dependence-direction matrix. For example, **map** operations have inherently parallel semantics, and, as such, one can freely interchange inwards the loops semantically corresponding to **map** operations (as long as the counts of the inner loops in the nest do not depend on the index of the **map**-like loop).

Let us analyze the legality of loop interchange for the three loop nests of our running example:

fig. 27(a): The direction matrix is `[=,=]` and, as such, it is legal to interchange the two loops, because it would result in direction matrix `[=,=]`. Moreover applying loop interchange in this case is highly beneficial because it ***optimizes locality of reference*** (in a CPU setting): the loop of index `i` appears in the innermost position after the interchange, which optimally exploits spatial locality for the write and read accesses to `A[j,i]`.

fig. 27(b): The direction matrices are $M = \begin{cases} \texttt{[<,<]} \\ \texttt{[=,<]} \end{cases}$ and $M^{intchg} = \begin{cases} \texttt{[<,<]} \\ \texttt{[<,=]} \end{cases}$ before and after interchange, respectively. It follows that the loop interchange is legal—because M^{intchg} satisfies theorem 7—and it also optimizes spatial locality (as before). What is interesting about this example is that after the interchange, ***the innermost loop has become parallel***, by theorem 6, because the outer loop caries all dependencies—the direction column corresponding to the outer loop consists only of < directions.

fig. 27(c): The direction matrix is `[<,>]` and ***interchanging the two loops is illegal*** because the direction matrix obtained after the interchange `[>,<]` starts with a > direction; this would mean that the current iteration depends on a future iteration, which is impossible, hence the interchange is illegal.

5.4 Loop Distribution: Legality and Applications

This section introduces a transformation, named loop distribution, that refers to the manner in which a loop can be safely distributed across its statements. Potential benefits are:

- loop distribution provides the bases for performing vectorization: the innermost loop is distributed across its TAC statements, and then the distributed loops are chunked (stripmined) by a factor that permits utilization of processor's vector instructions.
- loop distribution may enhance the degree of parallelism that can be statically mapped to the hardware, in a similar way in which it has been applied for flattening in section 4.2. There, a **map** was distributed across its statements such as to create perfect nests of parallel constructs, which are then flattened by applying corresponding re-write rules.

Loop distribution requires the construction of the dependency graph, which is defined below.

Definition 8 (Dependency Graph).
A dependency graph of a loop is a directed graph in which the nodes correspond to the statements of the loop nest and the edges correspond to dependencies. An edge is directed (points) from the source to the sink of the dependency, and is annotated with the direction corresponding to that dependence.

In the case when the loop contains another inner loop, then the inner loop is represented as a single statement that conservatively summarizes the behavior of all the statements of the inner loop.

The dependency graph of a loop can be used to characterize its parallel behavior:

Theorem 8 (Dependency Cycle).
A loop is parallel if and only if *its dependency graph does not have cycles.*

If the loop contains a cycle of dependencies, then it necessarily exhibits at least a cross iteration dependency (needed to form the cycle), and thus the loop is not parallel. The following theorem specifies how the transformation can be implemented:

Theorem 9 (Loop Distribution).
Distributing a loop across its statements can be performed in the following way:

1. *The dependency graph corresponding to the target loop is constructed.*
2. *The graph is decomposed into strongly-connected components (SCCs)*[11]*, and a new graph G' is formed in which the SCCs are nodes.*
3. *The loop can be safely distributed across its strongly-connected components, in the graph order of G'. Assuming a number k of SCCs, this means that the result of the transformation will be k loops, each containing the statements of the corresponding SCC. Inside an SCC, the statements remain in program order, but the distributed loops are ordered according to G'.*
4. *Array expansion must be performed for the variables that*
   - *are either declared inside the loop or overwritten in each iteration (output dependencies),* **and**
   - *are used in at least two strongly-connected components.*

The theorem above says that the statements that are in a dependency cycle must remain in (form) one loop (which is sequential by theorem 8). As such, the loop can be distributed across groups of statements corresponding to the strongly connected components (SCC) of the dependency graph. If the graph has only one SCC than it cannot be distributed. The resulting distributed loops are written in the order dictated by the graph of SCCs. We demonstrate theorem 9 on the simple code example presented below:

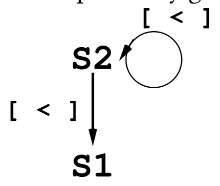
```
forall i = 2, N
S1:  A[i] = B[i-2] ...
S2:  B[i] = B[i-1] ...
endfor
```

The code has two dependencies:

$S_2 \rightarrow S_1$**:** In order for a dependency on B to exist the read from B in iteration i_1 of S_1 and the write to B in iteration i_2 of S_2 must refer to the same location. Hence $i_1 - 2 = i_2$, which means $i_1 > i_2$, hence S_2 is the source, S_1 is the sink and the direction vector is [<];

$S2 \rightarrow S2$**:** similarly, there is a dependency between the read from B in S_2 and the write to B in S_2 of direction vector [<].

The dependency graph is thus:



and it exhibits two strongly-connected components: one formed by statement S_2 and one formed by statement S_1. Loop distribution results in the following restructured code:

[11]A graph is said to be strongly connected if every vertex is reachable from every other vertex, i.e., a cycle. It is possible to find the strongly-connected components of an arbitrary directed graph in linear time $\Theta(V + E)$, where V is the number of vertices and E is the number of edges.

```
forall i = 2, N
S_2 :  B[i] = B[i-1] ...
endfor
forall i = 2, N
S_1 :  A[i] = B[i-2] ...
endfor
```

in which, according to the graph order, the loop corresponding to statement S_2 appears before the one corresponding to statement S_1. Please notice that this does not match the program order of statements S_1 and S_2 in the original program. Please also notice that the first loop is *not* parallel because the SCC consisting of S_2 has a (dependency) cycle, but the second loop is parallel because the SCC corresponding to S_1 does not have cycles.

One can notice that if a loop is parallel then it can be straightforwardly distributed across its statements in program order because:

- by theorem 8, the loop dependency graph have no cycles and thereby each statement is a strongly connected component;
- the program order naturally respects all dependencies.

Corollary 3 (Parallel Loop Distribution). *A parallel loop can be directly distributed across each one of its statements. The resulted loops appear in the same order in which their corresponding statements appear in the original loop.*

In fact, this is the theorem that provides the safety rationale for the **map** distribution used to perform flattening in section 4.2 (under the trivial observation that a **map** is a parallel loop).

Finally, it remains to demonstrate array expansion, mentioned in the fourth bullet of theorem 9. Assume the slightly modified code:

```
float tmp;
forall j = 0, N-1
     i = j + 2
S_1 :  tmp  = 2 * B[i-2]
S_2 :  A[i] = tmp
S_3 :  B[i] = tmp + B[i-1]
endfor
```

Statements S_1 and S_3 are in a dependency cycle, because there is a dependency $S_3 \rightarrow S_1$ with direction < caused by the write to and the read from array B, and a dependency $S_1 \rightarrow S_3$ with direction = caused by tmp. Statement S_2 is not in a dependency cycle, but there is a dependency $S_1 \rightarrow S_2$, and hence its distributed loop should follow the distributed loop containing S_1 and S_3. If we do not perform array expansion, the distributed code:

```
float tmp;
forall j = 0, N-1
     i = j + 2
S_1 :  tmp  = 2 * B[i-2]
S_3 :  B[i] = tmp + B[i-1]
endfor
forall j = 0, N-1
     i = j + 2
S_2 :  A[i] = tmp
endfor
```

does not respect the semantics of the original program because the second loop uses the same value of `tmp`—the one set by the last iteration of the first loop—while the original loop writes and then reads a different value of `tmp` for each iteration. It follows that we must perform array expansion for `tmp`, which means that we must expand it with an array dimension equal to the loop count and replace its uses with corresponding indexing expressions of the expanded array. This results in the following *correct* code:

```
float tmp[N];
forall j = 0, N-1
    i = j + 2
S1:  tmp[j]  = 2 * B[i-2]
S3:  B[i] = tmp + B[i-1]
endfor
forall j = 0, N-1
    i = j + 2
S2:  A[i] = tmp[j]
endfor
```

Array expansion requires to normalize the loop first—this means rewriting the loop such as its index starts from 0 and increases by 1 each iteration. This is why we have not written our example as **for** `i = 2, N+1`.

5.5 Eliminating False Dependencies (WAR and WAW)

Anti and output dependencies are often referred to as *false* dependencies because they can be eliminated in most cases by copying or privatization operations:

- Cross-iteration anti dependencies (WAR) typically correspond to a read from some original element of the array—whose value was set before the start of the loop execution—followed by an update to that element in a later iteration. As such, this dependency can be eliminated by copying (in parallel) the target array before the loop and rewriting the offending read access inside the loop such that it refers to the copy of the array.
- Cross-iteration output dependencies (WAW) can be eliminated by a technique named privatization (or renaming), whenever it can be determined that *every read access* from a scalar or array location *is covered by an update* to that scalar or memory location that was previously performed *in the same iteration*. Semantically, privatization moves the declaration of the offending variable inside the loop, because it has been already determined that the read/used value was produced earlier in the same iteration.
- Reasoning based on direction vectors is limited to relatively simple loop nests; for example it is difficult to reason about privatization by means of direction vectors.

5.5.1 Eliminating WAR Dependencies by Copying.

Consider the simple `C` code below which rotates an array in the right dimension by one:

```
float tmp = A[1];
for (int i=0; i<N-1; i++) {
    A[i] = A[i+1]; -- S1
}
A[N-1] = tmp;
```

The loop exhibits a cross-iteration anti dependency (WAR) $S_1 \rightarrow S_1$ (with direction vector [<]), and, as such, it is not safe to execute it in parallel. However, one can observe that the reads from `A` inside the loop correspond to the original elements of A before the loop, because they are rewritten in a later iteration. As such one can perform a copy of A before the loop, and replace the read access inside the loop to operate on the copy of array A. This preserves the original loop semantics and results in a parallel loop because the read and write accesses operate on different arrays, hence a dependency cannot occur. We present below the OpenMP code, which is a popular parallel API for multicore hardware:

```
float Acopy[N];
#pragma omp parallel for
for(int i=0; i<N; i++)} {
    Acopy[i] = A[i];
}
tmp = A[1];
#pragma omp parallel for
for (int i=0; i<N-1; i++) {
    A[i] = Acopy[i+1];
}
A[N-1] = tmp;
```

The code uses the `#pragma omp parallel for` annotation, by which the programmer solemnly swears that the following **for** loop is actually parallel; if the programmer breaks the vow then the parallel and sequential execution of the code will give different results; the API does not provide any guarantees about the absence of race-conditions. To compile with support for OpenMP annotation in gcc, please use the `-fopenmp` compilation flag. The number of cores that are going to be used for parallel execution can be set by environment variable `OMP_NUM_THREADS` for example by the command line `$export OMP_NUM_THREADS=8` if you would like to utilize eight threads.

5.5.2 *Eliminating WAW Dependencies by Privatization.*

Consider the contrived and ugly looking `C` code below:

```
int i;
int A[M];
for(i=0; i<N; i++){
  for(int j=0, j<M; j++) { -- writes slice A[0:M-1]
    A[j] = (4*i+4*j) % M;                          -- S1
  }
  for(int k=0; k<N; k++) { -- reads A[j] where j∈{0,...M-1}
    X[i,k] = X[i,k-1] * A[ A[(2*i+k)%M] % M];      -- S2
  }                        -- because % denotes modulus op
}
```

Analyzing the cross-iteration dependencies of the outer loop, one can observe that there are frequent output dependencies $S_1 \rightarrow S_1$ of all directions ($*$), because, in essence, all elements of `A` at indices $0 \ldots M-1$ are (over)written in each iteration of the outer loop. This also causes frequent cross-iteration WAR and RAW dependencies between S_1 and S_2 of all directions $*$ because S_2 reads some of the values of `A` which are written in S_1. The read access is also statically unanalyzable because the index into `A` depends on a value of `A` (i.e., it is an indirect-array access `A[ A[...] ]`).

It would thus seem that this is a hopeless case and parallel execution is a pipe dream. Not so! Actually the rationale of how to transform the outer loop into a parallel one is quite simple. One may observe that each iteration of the outer loop writes the same indices of `A`, namely the ones belonging to the closed integral interval `[0,M-1]`. One may also observe that S_2 reads from `A` elements whose indices necessarily belong to `[0,M-1]`—due to the two modulus-`M` operations. As such, one may conclude that any value read in S_2 must have been produced in the same iteration of the outer loop (in the inner loop enclosing S_1).

It follows that it is safe to rewrite the loop in the following way:

(1) declare a new variable `A'` of the same dimensions as `A` just inside the outer loop (or equivalently perform array expansion of array `A'` with a new outer dimension of size `N`), and
(2) replace all the uses of `A` in the outer loop by uses of `A'`;
- the resulting loop is safe to execute in parallel because there can be no dependencies on `A'` since each iteration uses a different array `A'`;

(3) as a last step, after the parallel execution of the loop terminates, one must copy (in parallel) the elements produced by the last iteration of the outer loop (i.e., `A'[0,...,M-1]` back to `A`.

The parallel OpenMP code that implements these steps is presented below:

```
int A[M];
int i;
#pragma omp for lastprivate(i) lastprivate(A)
for(i=0; i<N; i++) {
    for(int j=0, j<M; j++) {
        A[j] = (4*i+4*j) % M;
    }
    for(int k=0; k<N; k++) {
        X[i,k]=X[i,k-1] * A[ A[(2*i+k) % M] % M];
}   }
```

Declaring array `A` as private (by `private(A)`) would result in semantically performing steps (1) and (2) above. Declaring it as `lastprivate(A)` instructs the OpenMP compiler to also perform step (3)—to copy back the privately-maintained result of `A` of the last executing iteration into the globally-declared array `A`.

Please also note that the OpenMP execution will not allocate a new `A'` for each iteration of the outer loop—this is actually equivalent to performing array expansion which is also applicable here—but instead it will *allocate a copy of* `A` *for each active thread*, thus significantly reducing the memory footprint and/or the number of (de)allocations.

We also remark that `i` is also declared `lastprivate`—because its value might be needed after the loop. Semantically, `i` generates true cross-iteration dependencies, and we have said that those are difficult to eliminate in general, because they correspond to algorithmic properties. The technique by which variables such as `i`[12] can be resolved is not actually privatization (OpenMP uses the wrong nomenclature here), but rather induction-variable recognition and substitution.

We conclude by repeating that privatization can be applied whenever one can prove that every read access in an iteration is covered by a previously-performed write access in the same iteration. Privatization can be implemented by performing either array expansion or moving the declaration of the target variable from outside to inside the loop. However, it saves memory to allocate the private copy per active thread rather than per iteration, which is what OpenMP is doing. In the GPU context (CUDA) it is likely that you would have to implement privatization by array expansion.

5.6 Recognizing Data-Parallel Operators Hidden in Sequential, Imperative Code

Section 3 has argued that programs written with parallelism in mind should be built—or at least be reasoned about—in terms of a nested composition of operators that have inherently parallel semantics, such as **`map`**, **`reduce`**, **`scan`**, **`filter`**. An important question is then "How can one identify and extract such operators from a sequential (legacy) code base?". This section attempts to provide an (incomplete) answer to this question, in the simpler case when the hardware is a multi-core machine.

5.6.1 Recognizing Reduce Operators.

We say that a (scalar) variable `x` in a target loop is in a reduce pattern if and only if all the statements in which `x` appears are of the form `x = x ⊙ exp`, where `x` does not appears in `exp` and $\odot$ is an associative operator. Such statements naturally raise cross-iteration true dependencies (RAW). However, these dependencies can be resolved in a simple way, by:

(1) privatizing `x`, which is initialized with the neutral element for each thread;
(2) executing the loop in parallel, such that each thread computes its own partial value for `x`, corresponding to the iterations it executes;
(3) reducing the partial values of `x` across processors by a reduction tree—or sequentially if the number of cores is small, which is the case of multicores;
(4) adding to the result the value of `x` from before the loop.

This is a simple technique; we point to the work of Lu and Mellor-Crummey for advanced compiler algorithms for identifying and optimizing reductions [Lu and Mellor-Crummey. 1998].

In fact, the procedure above refers to parallelizing a **`map-reduce`** composition—see list homomorphism section 2—in which, intuitively, the various `exp`s for a given iteration are obtained and added together semantically by a **`map`** operation, and then reduced across iterations by $\odot$. When using OpenMP, one must remember that OpenMP supports a fixed (small) set of reduce operators,

[12]which are incremented by same ammount in each iteration of the loop

such as integral or float addition, multiplication, and perhaps taking the maximum/minimum; it is not possible to work with an arbitrary operator. Please also notice that the same kind of rationale can be used in principle for the case when the variable is of an array type—for example a reduce with vectorized addition—except that it will be more difficult to recognize the operator, since this will likely be implemented as a loop.

We demonstrate the case of reduction on the following code, where we aim to parallelize the outer loop:

```
int i, j;
float x = 6.0;
for(i=1; i<N; i++) {
    for(j=1; j<N; j++) {
        if ( A[i,j] >= 2.0 )     x += 2*A[i,j-1];
        else if( A[i,j] > 0.0 ) x += A[i-1,j+1];
    }
    if (i % (j+1) == 3)
        x += A[i,i];
}
```

One may observe that all uses of variable x inside the outermost loop respect the reduction pattern, i.e., x is only used inside reduction statements. The outer loop can be parallelized under OpenMP by placing the following reduction annotation just before the outermost loop:
`#pragma omp parallel for reduction(+:x) private(i,j)`
and the program can be compiled with gcc by using the `-fopenmp` flag.

The above imperative program is semantically equivalent with the following Futhark program

```
let x_ini = 6.0
let xs = map (+1) (iota (N-1)) |>
         map (\i ->
                let exps = map (+1) (iota (N-1)) |>
                    map (\j -> let a = A[i,j] in
                               if a >= 2.0   then 2*A[i,j-1]
                               else if a > 0 then A[i-1,j+1]
                               else 0.0
                        )
                let e = reduce (+) 0.0 exps
                let e' = if (i % (j+1) == 3) then A[i,i] else 0.0
                in  e + e'
             )
let x = (reduce (+) 0 xs)
in x_ini + x
```

in which, if desired, the inner parallelism can be either exploited or sequentialized.

Finally, we remark that imperative code patterns might be deceiving, for example a reduction pattern might actually turn to be a **`map`** or a **`scatter`**. For example, in the code below:

```
for(int i=0; i<N; i++)
   A[i] = A[i] + 2;
```

the statement that updates `A[i]` respects the reduction patterns, but the loop is actually a **map** operation: **map** (+2) A. This is a simple one, which is easy to recognize. But what about the following one:

```
for(int i=0; i<N; i++)
   A[ B[i] ] = A[ B[i] ] + 2;
```

The parallel semantics of the loop above actually depends on the contents of indirect array B, which is typically part of the dataset, and thus statically unknown:

1. if array B does not contains duplicated elements—i.e., $\forall i, j \in \{0 \ldots N-1\},\ i \neq j$ we have that B[i] $\neq$ B[j] then the loop has the semantics of a **scatter**:
   `scatter A B <| map (\k -> A[k]+2) B`.
2. if array B contains duplicated elements then the pattern is often called a generalized reduction on arrays (think histograms!), and can be executed in parallel for example by using an atomic add operation—this of course requires the operator to be associative and commutative, because the increments happen out of (the sequential) order.

The test whether B contains duplicates can also be performed in parallel, and may guard the two different versions: a sufficient condition for disproving duplicates is to check whether the values of B are strictly monotonically increasing; this can be computed with the following efficient predicate:
`map (+1) (iota (N-1)) |> map (\i -> B[i-1]<B[i]) |> reduce (&&) true`
A precise test is also possible by:
`N == (scatter (replicate N 0) B (replicate N 1) |> reduce (+) 0)`
which first writes 1 in an array of zeros at the indices of B, then sums the result and checks equality with the length of B. Here the idea is that if B contains duplicates than at least two ones will be written in the same location, and necessarily the sum will be smaller than N. However, this test is significantly slower (about 3×) than the monotonicity test.

5.6.2 *Recognizing Scan Operators.*

Scan operations are difficult to recognize by the compiler. In principle, a loop-based implementation of **scan** will result in dependency cycle indicating a true dependency (RAW) of distance one—i.e., the current iterations depends on the result of the previous one. However not all such dependency pattern indicate **scan** operations. Furthermore, **scan** can be sequentially implemented in a multitude of forms, and as such they are difficult pattern match.

For example, the sequential, imperative code below:

```
A[0] = B[0];
for(i=1; i<N; i++) {
    A[i] = A[i-1] + B[i];
}
```

is an inclusive scan: `let A = scan (+) 0 B`.

The following loop:

```
acc = 0;
for(i=0; i<N; i++) {
    acc = acc + i;
    A[i] = acc;
}
```

is also an inclusive scan, albeit applied to the `iota N` array: `let A = scan (+) 0 (iota N)`.

Let us conclude with a non-trivial example. What kind of scan is represented by the code below?

```
for(j=0; j<M; j++)
    A[0,j] = B[0,j];

for(i=1; i<N; i++) {
    for(j=0; j<M; j++)
        A[i,j] = A[i-1,j] + B[i,j];
}
```

One can observe that, if we transpose matrices A and B, then the inner loop will resemble an inclusive scan, because it will have the pattern: A^{tr}[j,i]=A^{tr}[j,i-1] + B[j,i]. It follows that the second loop nest (of depth 2) actually performs a **scan** on each column of the matrix B, and the first single loop nest initializes the first element of the column result with the corresponding element for B. The code is thus semantically equivalent to:

```
let A = transpose B |> map (scan (+) 0.0) |> transpose
```

5.6.3 *Recognizing Filter Operators.*

The natural implementation for `let B = filter pred A` is by a loop which selectively adds elements to the result array whenever the predicate succeeds on the current array element:

```
int k = 0;
for(i=0; i<N; i++) {
    if pred(A[i]) {
        B[k] = A[i];
        k = k + 1;
    }
}
```

This loop presents analysis challenges because:

(1) variable k does not respect the reduction pattern since it is read in B[k];
(2) variable k is incremented in only some of the loop iterations, hence it cannot be replaced by a closed-formed formula in i (i.e., k it is not a proper induction variable);
(3) this makes it difficult to prove (by the compiler) that accesses to B[k] cannot generate cross-iteration output dependencies.

Solutions exist to automatically parallelize such loops in which arrays are indexed based on such conditionally-incremented scalars—but they require *complex* analysis [Oancea and Rauchwerger 2015], which exploits the monotonic nature of the values of k in different iterations.

5.7 Loop Stripmining, Block and Register Tiling

This section discusses several simple compiler transformations that are going to be combined in various ways to optimize locality of reference (both temporal and spatial locality).

Stripmining refers to the following transformation, which is always safe to apply:

```
for(int i = 0; i<N; i++) {       for(int ii = 0; ii<N; ii+=T){
  iteration body              ⇒    for(int i=ii, i<min(ii+T,N); i++)
}                                       iteration body
                                 } }
```

In essence, a normalized loop is split into a perfect nest of two loops, in which the first loop goes with stride `T`, and the second one goes with stride `1`. Please notice that the resulting loop nest executes the same number of statements and in the same order as the original loop.

Block Tiling refers to the transformation that stripmines several consecutive innermost loops in a perfect loop nest—named $l_{k+1} \ldots l_{k+n}$—and then interchanges inwards the resulting loops of stride 1. The transformation is valid/safe if in the original program it is safe to interchange any of the loops l_{k+i}, $i \in \{1, \ldots, n-1\}$ in the innermost position. For example, the code below demonstrates block tiling a perfect loop nest of depth two:

```
for(i = 0; i<N; i++) {       for(ii=0; ii<N; ii+=T1) {
  for(j = 0; j<M; j++) {       for(jj=0; jj<M; jj+=T2) {
    iteration body       ⇒       for(i=ii; i<min(ii+T1,N); i++) {
  }                                for(j=jj; j<min(jj+T2,M); j++) {
}                                    iteration body
                             } } } }
```

Unroll and jam refers to the transformation that partially unrolls one or more of the outer loops in a perfect nest and then fuses (“jams”) the resulting loops. Equivalently, one can stripmine an outer loop, then interchange (distribute) it in the innermost position, then completely unroll it. The transformation is aimed at decreasing the number of memory loads and stores by storing to and reusing values from registers, and thus it is applied when the original loop nest contains data references that allow for temporal reuse—e.g., their indexes are invariant to some of the loops in the nest. Due to this, it is also known as “*register tiling*”. We demonstrate the transformation on the matrix-matrix multiplication code below:

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

The plan is to stripmine the loop of index j by a tile of size 2, and to interchange it to the innermost position, while performing the necessary loop distribution and array expansion:

```
for(i=0; i<N; i++) {
  for(jj=0; jj<M; jj+=2) {
    float cs[2];
    for(j=jj; j<min(jj+2,M); j++) {
        cs[j-jj] = 0.0;
    }
    for(k=0; k<N; k++) {
      for(j=jj; j<min(jj+2,M); j++) {
        cs[j-jj] += A[i,k] * B[k,j];
    } }
    for(j=jj; j<min(jj+2,M); j++) {
        C[i,j] = cs[j-jj];
    }
} }
```

One can observe that the access A[i,k] is invariant to its immediately contained loop of index j and thus it can be hoisted outside it and saved into a register. Then the loops of index j can be unrolled, and array cs can be scalarized as well:

```
for(i=0; i<N; i++) {
  for(jj=0; jj<M; jj+=2) {
    float c1, c2;
    if (jj   < M) c1 = 0.0;
    if (jj+1 < M) c2 = 0.0;
    for(k=0; k<N; k++) {
      float a;
      a = A[i,k];
      if (jj   < M) c1 += a * B[k,jj  ];
      if (jj+1 < M) c2 += a * B[k,jj+1];
    }
    if (jj   < M) C[i,jj  ] = c1;
    if (jj+1 < M) C[i,jj+1] = c2;
} }
```

In the resulted code, the accesses to the elements of A have been halved. We can similarly apply unroll and jam for the loop of index i with a tile size equal to 3. This will cut down the accesses to B by a factor of 3. The resulted code is shown in fig. 29.

```
for(ii=0; ii<N; ii+=3) {
  for(jj=0; jj<M; jj+=2) {

    float c11, c12, c21, c22, c31, c32;

    if (ii   < N && jj   < M) c11 = 0.0;
    if (ii+1 < N && jj   < M) c21 = 0.0;
    if (ii+2 < N && jj   < M) c31 = 0.0;
    if (ii   < N && jj+1 < M) c12 = 0.0;
    if (ii+1 < N && jj+1 < M) c22 = 0.0;
    if (ii+2 < N && jj+1 < M) c32 = 0.0;

    for(k=0; k<N; k++) {

      float a1, a2, a3, b1, b2;

      if (ii   < N) a1 = A[ii  ,k];
      if (ii+1 < N) a2 = A[ii+1,k];
      if (ii+2 < N) a3 = A[ii+2,k];
      if (jj   < M) b1 = B[k,jj  ];
      if (jj+1 < M) b2 = B[k,jj+1];

      if (ii   < N && jj   < M) c11 += a1 * b1;
      if (ii+1 < N && jj   < M) c21 += a2 * b1;
      if (ii+2 < N && jj   < M) c31 += a3 * b1;
      if (ii   < N && jj+1 < M) c12 += a1 * b2;
      if (ii+1 < N && jj+1 < M) c22 += a2 * b2;
      if (ii+2 < N && jj+1 < M) c32 += a3 * b2;
    }

    if (ii   < N && jj   < M) C[ii,  jj  ] = c11;
    if (ii+1 < N && jj   < M) C[ii+1,jj  ] = c21;
    if (ii+2 < N && jj   < M) C[ii+2,jj  ] = c31;
    if (ii   < N && jj+1 < M) C[ii,  jj+1] = c11;
    if (ii+1 < N && jj+1 < M) C[ii+1,jj+1] = c21;
    if (ii+2 < N && jj+1 < M) C[ii+2,jj+1] = c31;
  }
}
```

Fig. 29. Result of unroll-and-jam applied to matrix-matrix multiplication, where the first and second outer loops were tiled with sizes 3 and 2, respectively. The number of accesses to A and B has been reduced by a factor of 2× and 3×, respectively, at the expense of introducing some conditional statements.

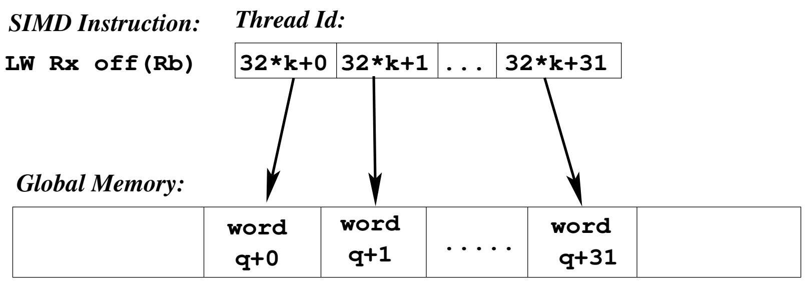


Fig. 30. Coalesced Access Pattern: the threads executing in lockstep access consecutive global-memory locations in a SIMD load or store instruction.

6 APPLICATIONS OF DATA-DEPENDENCE ANALYSIS ON GPU HARDWARE

This section discusses in the context of GPGPU-hardware execution, how to apply tiling transformations in order to optimize spatial and temporal locality of reference:

- section 6.1 introduces the pattern of "coalesced-access" to global memory—which is efficiently supported by GPU hardware—and discusses how transposition can be used to transform non-coalesced accesses to coalesced ones in many cases.
- section 6.2 presents the typical GPU implementation of matrix transposition, which uses block tiling to ensure coalesced access to global memory for both read and write operations;
- section 6.3 starts from the naive implementation of matrix-matrix multiplication and demonstrates how block tiling can be applied to derive an optimized implementation that exploits temporal locality—which is implemented by utilizing GPU's scratchpad memory as a software-managed cache;
- section 6.4 proposes an exercise in which the reader is directed to follow the steps necessary to combine register and block tiling in order to further optimize locality of reference in the case of matrix-matrix multiplication executed on GPU hardware.

6.1 Optimizing the Spatial Locality of Read/Write Accesses on GPUs by Transposition

Definition 9 (Coalesced Access).
A read or write access to GPU global memory is said to be coalesced, if the consecutive threads that execute in lockstep access consecutive global-memory locations in the corresponding SIMD load or store instruction.

The definition above and fig. 30 introduce the notion of coalesced access to global memory. This corresponds to a certain memory pattern, whose spatial locality is efficiently supported by the GPU hardware. In essence if the threads that execute in lockstep access consecutive memory locations in their SIMD load or store instruction, then the GPU memory system will perform the data transfer in only one memory transaction. Otherwise, if the memory locations accessed by the w lockstep-executing threads are spread out in memory, executing the SIMD load or store instruction may require up to w memory transactions. Given that, in practice, common values for w are 16 and 32, optimizing a program to perform coalesced (rather than uncoalesced) accesses to global memory gives an important (huge) performance boost.

Luckily, in the case when the array subscript is an affine formula of the indices of the enclosing loop nest, then a general technique exists for optimizing coalesced accesses. The technique relies on changing the layout of the corresponding multi-dimensional arrays by means of (generalized) transposition. We demonstrate the technique on the code example below:

```
float A[N,64];
// ... code to fill in array A
float B[N,64];
forall (i=0; i<N; i++) {  // parallel
  float tmpA, tmpB, accum;
  tmpB = A[i,0] * A[i,0];
  B[i,0] = tmpB;
  for(j=1; j<64; j++) { // sequential
    tmpA   = A[i,j];
    accum  = sqrt(tmpB) + tmpA*tmpA;
    B[i,j] = accum;
    tmpB   = accum;
  }
}
```

In the code example, the outermost loop is parallel (why?) and the inner loop is sequential (why?). It follows that the CUDA kernel would correspond to the body of the outermost loop (which includes the inner loop), and a number of (at least) N CUDA threads are spawned to execute the kernel.

Let us analyze what happens when a warp[13] of threads execute statement `tmpA = A[i,j]` in some fixed iteration `j` of the inner loop. Since `i` ranges through thread numbers, the values of `i` for the current warp `k` can be written as `32*k+0, 32*k+1, ..., 32*k+31`. It follows that in the original program, the SIMD load instruction from array `A` will read the following (flattened) words/locations of `A`: `(32*k+0)*64 + j, (32*k+1)*64 + j, ..., (32*k+31)*64 + j`. This corresponds to a strided access in which the stride is 64 words, no matter of the value of `j`—i.e., each thread accesses a location which is 64 words apart from the previous thread access. As such we can expect that our SIMD load instruction will generate 32 independent memory transactions, resulting in terribly inefficient execution. Similar thoughts apply to the SIMD instructions executing the load from `A[i,0]` and the store to `B[i,0]` and `B[i,j]`.

One can optimize the program by changing the layout of the input and result arrays `A` and `B`, respectively:

(1) introduce a new computation before the loop that stores in array `A'` the transposed form of array `A`;
(2) inside the loop, rewrite the uncolaesced accesses to `A` and `B` into coalesced accesses to `A'` and `B'`—where `B'` is similarly the transpose of `B`;
(3) introduce a new computation after the loop that transposes `B'` into `B`.

Applying the transformation results into the semantically-equivalent program:

[13]In CUDA terminology, a warp is the group threads that execute in lockstep an SIMD instruction.

```
float A [N,64];
float A'[64,N];
// ... code to fill in array A
float B [N,64];
float B'[64,N];
A' = transpose(A);
forall (i=0; i<N; i++) {  // parallel
  float tmpA, tmpB, accum;
  tmpB = A'[0,i] * A'[0,i];
  B'[0,i] = tmpB;
  for(j=1; j<64; j++) { // sequential
    tmpA   = A'[j,i];
    accum  = sqrt(tmpB) + tmpA*tmpA;
    B'[j,i]= accum;
    tmpB   = accum;
  }
}
B = transpose(B');
```

Now all the read and write accesses to arrays `A’` and `B’` are coalesced. Take for example the read access to `A’[j,i]`. The current warp `k` will result in the following values of `i`: `32*k+0, 32*k+1, ..., 32*k+31`. For a fixed `j`, the flat locations read from `A’[j,i]` by the current warp will be `j*N + (32*k+0), j*N + (32*k+1), ..., j*N + (32*k+31)`, because the size of the innermost dimension of `A’` is `N`. It follows that the current warp accesses in the same SIMD load instruction consecutive memory locations, hence the access is coalesced. Similar thoughts apply to the read from `A’[0,i]` and to the write to `B’[0,i]` and `B’[j,i]`.

Note that the original program performs `N*64` reads (from `A`) and `N*64` writes (to `B`) to global memory. The transformed program performs 3× more reads and writes than the original program, because of the two `transpose` operations. In spite of this, one is likely to observe that the transformed program executes much faster than the original one, because it exhibits the kind of spatial locality (of accesses to `A’` and `B’`) that is efficiently supported by the GPU hardware.

Next section presents how the `transpose` operation can be efficiently implemented in CUDA, such that it uses only coalesced (read and write) accesses to global memory.

6.2 Transposition: Block Tiling Optimizes Spatial Locality

In the previous section we have seen how one can transform uncoalesced accesses to global memory into coalesced ones by means of transposition. However, this assumes that the `transpose` operation itself can be written only in terms of coalesced accesses to global memory. This step is nontrivial and is covered in this section.

We start with the mathematical definition of the `transpose` operator: given a $r \times c$ matrix `A`—where r and c denotes the number of rows and columns, respectively—the transpose of `A`, denoted `A’` is a $c \times r$ matrix such that `A’[j,i] = A[i,j]`, $\forall i \in \{0 \ldots r-1\}$ and $\forall j \in \{0 \ldots c-1\}$. This leads to the following naive code:

```
forall (i=0; i<R; i++) {  // parallel
  forall (j=0; j<C; j++) { // parallel
    A'[j,i] = A[i,j];
} }
```

In the naive code, both loops are parallel and the second one of index `j` iterates faster. It follows that the read access from `A[i,j]` will be coalesced and the write access to `A'[j,i]` will be uncoalesced. Intuitively, if `C` is a multiple of 32, then:

- a warp of threads would correspond to the same value of `i` and consecutive values of `j`: `k*32+0, k*32+1, ..., k*32 + 31`,
- the flat indices of `A'[j,i]` written by a warp in an SIMD instruction will have the form: `(k*32+0)*R + i, (k*32+1)*R + i, ..., (k*32+31)*R + i` which corresponds to a strided access with stride equal to the number of rows `R`, hence the write is not coalesced.

We aim to have both accesses to `A` and `A'` in coalesced form. The first step is to apply block tiling with a generic tile of size `T` to both parallel loops. We recall that block tiling corresponds to stripmining both loops with an inner one that goes with a count `T` and increment one, followed by interchanging the stripmined loops in the innermost position. The safety of the interchange is guaranteed by the parallel nature of the two loops (see corollary 2). After applying block tiling the code becomes:

```
forall (ii=0; ii<R; ii+=T) {  // parallel grid.y
  forall (jj=0; jj<C; jj+=T) { // parallel grid.x
    forall (i=ii; i<min(ii+T,R); i++) {  // parallel block.y
      forall (j=jj; j<min(jj+T,R); j++) {  // parallel block.x
        A'[j,i] = A[i,j];
    } }
} }
```

In the code above, the outer two loops of indices `ii` and `jj` will correspond to the CUDA grid, while the innermost two loops of indices `i` and `j` will correspond to the CUDA block—i.e., we will work with two-dimensional grids of blocks, in which the block is also two dimensional. Please note that the total size of a block cannot exceed 1024 in CUDA, and the size of our CUDA block will be `T`×`T` because each inner loop has count `T`. It follows that $T^2 \leq 1024$, hence valid values of `T` are less than or equal to 32.

We assume for simplicity that `R` and `C` evenly divide `T`. Let us compute first the indices that are read from and written into arrays `A` and `A'`, respectively, for a given block `[ii,jj]` in the grid. The innermost two loops have indices `i = ii+0,...,ii+T-1` and `j = jj+0,..., jj+T-1`, hence our block will read the slice `A[ii:ii+T, jj:jj+T]`. By convention the slice excludes the last element, so the size of the slice is naturally T^2. We would like to read the slice in shared memory—we recall that CUDA's "shared" memory refers to fast scratchpad memory—and figure out from there how to rewrite the write access to be coalesced. The CPU orchestrating code is:

```
void transposeFloat( float* mat, float* mat_tr,
                     const unsigned int R, // height
                     const unsigned int C  // width
) {
  unsigned int dimy = (R + T - 1) / T;
  unsigned int dimx = (C + T - 1) / T;
  dim3 block(T,T,1), grid (dimx, dimy, 1);
  transpose<<< grid, block >>>(mat, mat_tr, R, C);
}
```

and the intermediate code of the CUDA kernel is presented below:

```
__global__ void transpose(float* A, float* trA, int R, int C) {
  __shared__ float tile[T][T];
  unsigned int tidx = threadIdx.x;
  unsigned int tidy = threadIdx.y;
  unsigned int j = blockIdx.x*T + tidx;
  unsigned int i = blockIdx.y*T + tidy;
  if( j < C && i < R )
    tile[tidy][tidx] = A[i*colsA + j];
  __syncthreads();
  if ( j < C && i < R )
    trA[j*R + i] = tile[tidy][tidx];
}
```

The code above simply spawns a CUDA thread for each of the elements of A, then it computes the corresponding indices i and j for a given thread. Please note that the iteration space corresponds directly to the input array: the two-dimensional grid contains $\lceil\frac{R}{T}\rceil \times \lceil\frac{C}{T}\rceil$ blocks, each block containing T×T elements. Index i is obtained by computing the displacement of the current block on the y axis—i.e., blockIdx.y*T to which we add the row number of the current block—i.e., threadIdx.y. Index j is computed in a similar way. The kernel then copies element A[i,j] in a T×T array, named tile, which is allocated in shared memory, and finally it reads the same element from shared memory and writes it in the transposed position in the transposed array.

So far, we have accomplished nothing, because the access to trA is still uncoalesced; in essence a coalesced access should have a + tidx term, but our access has a + tidx*R term which would correspond to a strided access to global memory by a stride equal to R. This is because we currently use thread [tidy,tidx] to write the local element tile[tidy][tidx] to result array trA.

The essential step in fixing the uncoalesced access is to use the current thread [tidy,tidx] to write the "transposed" local element in the current block, i.e., tile[tidx][tidy]. This introduces non-coalesced access to tile, which is fine because tile is allocated in shared memory and it does not suffer the penalty of non-coalesced accesses (only global-memory accesses do!). The global index in A corresponding to local element tile[tidx][tidy] will be i' = blockIdx.y*T + tidx and j' = blockIdx.x*T + tidy, which will be written in trA[j',i'], resulting in coalesced access to trA because i' carries the + tidx term—i.e., T consecutive threads on the x direction will write T consecutive element locations in memory. The final code is presented below:

```
__global__ void transpose(float* A, float* trA, int R, int C) {
  __shared__ float tile[T][T+1];
  unsigned int tidx = threadIdx.x;
  unsigned int tidy = threadIdx.y;
  unsigned int j = blockIdx.x*T + tidx;
  unsigned int i = blockIdx.y*T + tidy;
  if( j < C && i < R )
    tile[tidy][tidx] = A[i*colsA + j];
  __syncthreads();
  unsigned int j' = blockIdx.x*T + tidy;
  unsigned int i' = blockIdx.y*T + tidx;
  if ( j' < C && i' < R )
    trA[j'*R + i'] = tile[tidx][tidy];
}
```

The observant reader has noticed a modification in the declaration of the shared memory buffer `tile`: previously it was declared as a `T×T` array, while now it is declared as a `T×(T+1)` array. The reason for that is that, in CUDA, the number of shared-memory banks is a power of two: typically 16 or 32. The shared memory does not need to be accessed in coalesced fashion, but it suffers a significant performance bottleneck when two threads in the same warp access in an SIMD load or store instruction two memory locations situated on the same bank. In this case the two (or multiple) accesses are sequentialized; while the shared-memory transactions have much smaller latency than global-memory transactions, this bottleneck may still significantly restrict performance gains.

Assume that in our code we use `T=32` and CUDA shared memory is also distributed on 32 banks. If we declare our `tile` array to be of size `T×T` then the last read access from `tile[tidx][tidy]` would have an entire warp reading different memory locations from the same memory bank. This is because `tidx` ranges through `0,...,31` and `tidy` is the same for an entire warp—i.e., with our setting we will have `tidy` ranging through the warps in a CUDA block. It follows that an entire warp will access the same shared-memory bank `tidy` in the last read from `tile[tidx][tidy]`, and the `32` shared-memory transactions of the warp will be sequentialized. Luckily, the fix is simple: we make the innermost dimension of our shared-memory buffer equals to `T+1` rather than `T`, which will spread the accesses in a warp to different banks. This comes at the expense of allocating an additional number of `T` shared-memory locations which will not be utilized.

6.3 Matrix-Matrix Multiplication: Block Tiling Optimizes Temporal Locality

We turn our attention to how block tiling can optimize temporal locality of reference. We use for demonstration the dense matrix-matrix multiplication algorithm. We start from the naive implementation given below, which multiplies matrices `A` and `B` whose sizes are `M×U` and `U×N`, respectively, and results in a matrix `C` of size `M×N` (i.e., `M` rows and `N` columns):

```
forall (i=0; i<M; i++) {  // parallel map
  forall (j=0; j<N; j++) {  // parallel map
    float c = 0.0;
    for (k=0; k<U; k++) {    // sequential (reduce o map)
      c += A[i,k] * B[k, j]
    }
    C[i,j] = c;
  }
}
```

One can observe that there is potential to exploit temporal locality: read access `A[i,k]` is invariant to the second loop of index `j`, and read access `B[k,j]` is invariant to the outermost loop of index `i`. As such the program redundantly reads each element of `A` `N` times, and each element of `B` `M` times. However, in the current form it is not possible to (fully) exploit the temporal locality to arrays `A` and `B`.

We restructure the code by applying block tiling to the two outer loops—i.e., we stripmine them, each with a tile equal to `T` and we interchanges inwards the stripmined loops of stride one—and we also stripmine the innermost loop of index `k`. This results in the code shown in fig. 31.

In the restructured code, we reason as before that the two outer loops correspond to a two-dimensional CUDA grid containing $\lceil\frac{M}{T}\rceil \times \lceil\frac{N}{T}\rceil$ two-dimensional blocks. A CUDA block would correspond to the third and fourth loops of indices `i` and `j`, thus each block contains $T \times T$ threads. With the code in this form we turn our attention to the elements that are accessed in the innermost

```
forall (ii=0; ii<M; ii+=T) {  // parallel grid.y
  forall (jj=0; jj<N; jj+=T) {  // parallel grid.x
    forall (i=ii; i<min(ii+T,M); i++) { // parallel block.y
      forall (j=jj; j<min(jj+T,M); j++) { // parallel block.x
        float c = 0.0;
        for (kk=0; kk<U; kk+=T) {    // sequential (reduce o map)
          // The loop below reads the following slices:
          // A[ii:ii+T, kk:kk+T] and B[kk:kk+T, jj:jj+T]
          for (k=kk; k<min(kk+T,U); k++) { // sequential
            c += A[i,k] * B[k, j]
        } }
        C[i,j] = c;
    } }
} }
```

Fig. 31. C-like pseudocode for block-tiled matrix matrix multiplication

loop of index k. This loop accesses elements of A in the slice A[ii:ii+T, kk:kk+T] and elements of B in the slice B[kk:kk+T, jj:jj+T].

In essence, a given CUDA block of size T×T, performs in some iteration of the kk loop a total of 2×T^3 accesses to global memory (T^3 to array A and T^3 to array B), but the number of *distinct* elements of A and B which are accessed in a block is only T^2. We can take advantage by this property by making the threads of a CUDA block to collectively copy just before the entry into the loop of index k the slices A[ii:ii+T, kk:kk+T] and B[kk:kk+T, jj:jj+T] to buffers allocated in shared memory. This would allow the innermost loop to only access the shared memory buffers instead of global memory. The resulted CUDA kernel code is shown in fig. 32.

The CPU orchestrating code that calls the kernel is shown below:

```
double MatMatMultFloat( float* A_d, float* B_d, float* C_d,
                        int M, int N, int U ) {
  unsigned int dimy = (M + T - 1) / T;
  unsigned int dimx = (N + T - 1) / T;
  dim3 block(T,T,1), grid (dimx, dimy, 1);

  unsigned long int elapsed;
  struct timeval t_start,t_end,t_diff;
  gettimeofday(&t_start, NULL);
  MatMatMult<<< grid, block >>>(A_d, B_d, C_d, M, N, U);
  gettimeofday(&t_end, NULL);
  timeval_subtract(&t_diff, &t_end, &t_start);
  elapsed=(t_diff.tv_sec*1e6 + t_diff.tv_usec);
  double flops = 2.0 * M * N * U;
  double gigaFlops=(flops*1.0e-3f) / elapsed;
  return gigaFlops;
}
```

```
__global__ void MatMatMult( float* A, float* B, float* C
                          , int M, int N, int U) {
  __shared__ float Ash[T][T];
  __shared__ float Bsh[T][T];
  unsigned int tidx = threadIdx.x, tidy = threadIdx.y;
  unsigned int ii = blockIdx.y * T, jj = blockIdx.x * T;
  unsigned int i  = ii + tidy;
  unsigned int j  = jj + tidx;
  float c = 0.0;
  for (int kk=0; kk<U; kk+=T) {     // sequential (reduce o map)
    Ash[i-ii][tidx] = (i < M && kk+tidx < U) ?
                      A[i*U + (kk+tidx)] : 0.0;
    Bsh[tidy][j-jj] = (j < N && kk+tidy < U) ?
                      B[(kk+tidy)*N + j] : 0.0;
    __syncthreads();
    #pragma unroll
    for (int k=0; k<T; k++) { // sequential
      c += Ash[i-ii][k] * Bsh[k][j-jj];
    }
    __syncthreads();
  }
  if (i < M && j < N)
    C[i*N + j] = c;
}
```

Fig. 32. CUDA kernel code for block-tiled matrix matrix multiplication

The CPU code measures the kernel runtime, and then computes the performance in giga floating-point operations per second (1 GFlops per sec = 10^9 flops per sec). The number of floating-point operations for matrix-matrix multiplication is: 2×M×N×U because there are three nested loops of size M, N, and U which perform a floating-point addition and a multiplication. The `elapsed` time has been computed in microseconds (1 microsecond equals 10^{-6} seconds). It follows that the number of giga-flops per second equals the number of flops divided by the time in microseconds and the result is divided by an extra 10^3.

We conclude by remarking that block tiling can be similarly applied in one dimension or in more dimensions (e.g., three dimensions)—it all depends on how many loops do we need to tile in order to exploit temporal locality.

```
forall (i=0; i<M; i++) {  // parallel
    float s = 0, a = A[i];   float a_sq = a * a;
    for (k=0; k<N; k++) {    // sequential
      float a_k = A[k];
      s += sqrt (a_sq - a_k*a_k);
    }
    B[i] = s;
}
```

For example, in the code above the access to A[k] is invariant to the outermost parallel loop, and can be optimized by one-dimensional tiling as below:

```
forall (ii=0; ii<M; ii+=T) {  // grid.x
  forall (i=ii; i<min(ii+T,M); i++) { // block.x of size T
    __shared__ float Ash[T];
    float s = 0, a = A[i];
    float a_sq = a * a;
    for (kk=0; kk<N; kk+=T) {  // sequential
      // collectively copy A[kk:kk+T] in Ash[0:T]
      for (k=kk; k<min(kk+T,N); k++) { // sequential
        float a_k = Ash[k-kk];
        s += sqrt (a_sq - a_k*a_k);
      }
    }
    B[i] = s;
  }
}
```

6.4 Exercise: Block and Register Tiling for Matrix-Matrix Multiplication

This section presents the steps by which block and register tiling can be combined to further optimize the temporal locality of matrix-matrix multiplication. We start as before with the naive code:

```
forall (i=0; i<M; i++) {  // parallel map
  forall (j=0; j<N; j++) {  // parallel map
    float c = 0.0;
    for (k=0; k<U; k++) {    // sequential (reduce o map)
      c += A[i,k] * B[k, j]
    }
    C[i,j] = c;
  }
}
```

We then tile the three loops in the following way:

- we stripmine the parallel dimension of index i by a tile of size T and stride 1;
- we double stripmine the second parallel dimension of index j by a tile of size T^2 and stride T, and then by a tile of size T and stride 1;
- we stripmine the sequential dimension of index k by a tile of size T and stride 1.

Since the loops of index i and j are parallel, we can interchange their tiles inwards and distribute them as we desire, and our desire is as follows:

- we move the tile of loop i innermost and we sequentialize it;
- we move the double tiles of the loop of index j just inside the original loop of index k.

The resulting code is shown in fig. 33:

- the two outer parallel loops of indices ii and jjj correspond to the two-dimensional CUDA grid, which contains $\lceil \frac{M}{T} \rceil \times \lceil \frac{N}{T^2} \rceil$ blocks;

```
unsigned int ii, i, jjj, jj, j, kk, k;
forall (ii = 0; ii < M; ii += T ) {           // parallel grid.y
  forall (jjj = 0; jjj < N; jjj += T*T ) {    // parallel grid.x
    float cs[T][T][T];

    forall(jj=jjj; jj<min(jjj+T*T,N); jj+=T){ // parallel block.y
      forall(j=jj; j<min(jj+T,N); j++) {      // parallel block.x
        for (i=ii; i<min(ii+T,M); i++) {      // sequential
          cs[(jj-jjj)/T][j-jj][i-ii] = 0.0;
    } } }

    for (kk = 0; kk < U; kk += T) {                 // sequential
      // here we will insert a collective copy to shared
      // memory of the slice: A[ii : ii+T, kk : kk+T]
      for (k = kk; k < min(kk+T,U); k++) {          // sequential
        forall (jj=jjj; jj<min(jjj+T*T,N); jj+=T) { // block.y
          forall (j=jj; j<min(jj+T,N); j++) {       // block.x
            float b = B[k,j];                       // hoisted out
            for (i = ii; i < min(ii+T,M); i++) {    // sequential
              // please modify here to read from shared
              //  memory Ash and to scalarize cs
              cs[(jj-jjj)/T][j-jj][i-ii] += A[i,k] * b;
    } } } } }

    forall (jj=jjj; jj<min(jjj+T*T,N); jj+=T) {     // block.y
      forall (j=jj; j<min(jj+T,N); j++) {           // block.x
        for (i = ii; i < min(ii+T, M); i++) {       // sequential
          C[i,j] = cs[(jj-jjj)/T][j-jj][i-ii];
    } } }
}  }
```

Fig. 33. C-like pseudocode for block+register tiled matrix matrix multiplication

- the parallel loops of indices `jj` and `j` correspond to the CUDA block, which, as before, has size T×T;
- the loop of index `i` is sequentially executed (part of the kernel code);
- we have explicitly distributed the loops of indices `jj`, `j` and `i` across the intialization of `c=0.0`, the computation of `c` and the update of result matrix `C`. This has required to expand scalar `c` with three array dimensions all equal with `T`. This step was not shown in the block-tiled matrix matrix multiplication because there the distributed loops were only the ones forming the CUDA block. These two dimensions of size `T` can be ignored as well in the CUDA code for the same reason. However, `c` needs to be expanded with a dimension of size `T` in the CUDA code due to the distribution of the loop of index `i`, which is sequentialized in the block+register tiled version of the code;

- please notice that the read from global memory `B[k,j]` has been hoisted outside the innermost loop of index `i` since it is invariant to it;
- each thread now computes `T` elements on a column of the result array: the local thread of block index `threadIdx.y = (jj-jjj)/T` and `threadIdx.y = j-jj` in some block `blockIdx.y = ii` and `blockIdx.x = jjj]` computes the elements `C[ii:ii+T,j]` by sequentially iterating at most `T` times through the loop of index `i` in the last loop nest.

Your task is to implement the CUDA code for the block+register tiled matrix matrix multiplication—including the CPU orchestration code and the kernel code—specialized for the case when `T=16`. The following details are probably important:

- make the threads of the block to collectively copy the slice `A[ii:ii+T, kk:kk+T]` into shared memory just inside the loop of index `kk`;
- Insert the necessary synchronization and replace the read from `A[i,k]` in the innermost loop body with a read access from shared memory;
- semantically, each thread should work with its own (private) array `cs` of size `T`; you do not necessarily need to scalarize this array since the CUDA compiler might do it for you. However, if in doubt you may scalarized the `cs` array for `T=16`—please consult section 5.7 and fig. 29 for inspiration.
- please make sure that all the loops of indices `i` and `k` are normalized—i.e., they go from `0` to `T` with a stride of `1`—and the loop of index `i` is unrolled (specified with `#pragma unroll` before it), but the loop of index `k` is not unrolled!

Please report the GFlops per second achieved by your implementation, and compare the performance with the provided block-tiled version. Please also answer what is the degree of temporal reuse for the matrix `A` and for the matrix `B` in your implementation, i.e., “A read from `A` and `B` stored in global memory is amortized by how many shared-memory or register accesses?”

REFERENCES

Lars Bergstrom, Matthew Fluet, Mike Rainey, John Reppy, Stephen Rosen, and Adam Shaw. 2013. Data-only Flattening for Nested Data Parallelism. In *Procs. of the 18th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP '13)*. ACM, New York, NY, USA, 81–92. https://doi.org/10.1145/2442516.2442525

Lars Bergstrom and John Reppy. 2012. Nested Data-parallelism on the GPU. *SIGPLAN Not.* 47, 9 (Sept. 2012), 247–258. https://doi.org/10.1145/2398856.2364563

Guy E. Blelloch. 1989. Scans as Primitive Parallel Operations. *Computers, IEEE Transactions* 38, 11 (1989), 1526–1538.

Guy E. Blelloch. 1996. Programming Parallel Algorithms. *Communications of the ACM (CACM)* 39, 3 (1996), 85–97.

Guy E Blelloch, Jonathan C Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. 1994. Implementation of a Portable Nested Data-Parallel Language. *Journal of parallel and distributed computing* 21, 1 (1994), 4–14.

Robert Clifton-Everest, Trevor L. McDonell, Manuel M. T. Chakravarty, and Gabriele Keller. 2017. Streaming Irregular Arrays. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell (Haskell 2017)*. ACM, New York, NY, USA, 174–185. https://doi.org/10.1145/3122955.3122971

Murray Cole. 1993. Parallel Programming, List Homomorphisms and the Maximum Segment Sum Problem. In *Procs. of Parco 93*.

Michel Dubois, Murali Annavaram, and Per Stenstrm. 2012. *Parallel Computer Organization and Design*. Cambridge University Press, New York, NY, USA.

Jeremy Gibbons. 1996. The Third Homomorphism Theorem. *Journal of Functional Programming (JFP)* 6, 4 (1996), 657–665.

Sergei Gorlatch. 1996. Systematic Extraction and Implementation of Divide-and-Conquer Parallelism. In *PLILP'96*. 274–288.

Mary W. Hall, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, and Monica S. Lam. 2005. Interprocedural Parallelization Analysis in SUIF. *Trans. on Prog. Lang. and Sys. (TOPLAS)* 27(4) (2005), 662–731.

Troels Henriksen, Ken Friis Larsen, and Cosmin E. Oancea. 2016. Design and GPGPU Performance of Futhark's Redomap Construct. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY 2016)*. ACM, New York, NY, USA, 17–24.

Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Ben Lippmeier, and Simon Peyton Jones. 2012. Vectorisation Avoidance. In *Proceedings of the 2012 Haskell Symposium (Haskell '12)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/2364506.2364512

Ken Kennedy and John R Allen. 2001. Optimizing compilers for modern architectures: a dependence-based approach. (2001).

Rasmus Wriedt Larsen and Troels Henriksen. 2017. Strategies for Regular Segmented Reductions on GPU. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing (FHPC 2017)*. ACM, New York, NY, USA, 42–52. https://doi.org/10.1145/3122948.3122952

B. Lu and J. Mellor-Crummey. 1998. Compiler Optimization of Implicit Reductions for Distributed Memory Multiprocessors. In *Int. Par. Proc. Symp. (IPPS)*.

Frederik M Madsen and Andrzej Filinski. 2016. Streaming nested data parallelism on multicores. In *Proceedings of the 5th International Workshop on Functional High-Performance Computing*. ACM, 44–51.

Sungdo Moon and Mary W. Hall. 1999. Evaluation of Predicated Array Data-Flow Analysis for Automatic Parallelization. In *Int. Symp. Princ. and Practice of Par. Prog. (PPoPP)*. 84–95.

G. E. Moore. 2006. Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter* 11, 3 (Sept 2006), 33–35. https://doi.org/10.1109/N-SSC.2006.4785860

Cosmin E. Oancea and Lawrence Rauchwerger. 2011. A Hybrid Approach to Proving Memory Reference Monotonicity. In *Procs. Int. Lang. Comp. Par. Comp. (LCPC)*.

Cosmin E. Oancea and Lawrence Rauchwerger. 2012. Logical Inference Techniques for Loop Parallelization. In *Procs. of Int. Conf. Prog. Lang. Design and Impl. (PLDI)*. 509–520.

Cosmin E. Oancea and Lawrence Rauchwerger. 2015. Scalable Conditional Induction Variables (CIV) Analysis. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '15)*. IEEE Computer Society, Washington, DC, USA, 213–224. http://dl.acm.org/citation.cfm?id=2738600.2738627

John Reppy and Nora Sandler. 2015. Nessie: A NESL to CUDA Compiler. Presented at the *Compilers for Parallel Computing Workshop (CPC '15)*. (Jan. 2015). Imperial College, London, UK.