# Programming the Greedy CAM Machine

Erik Ruf

January 26, 2007

MSR-TR-2007-04

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

# Programming the Greedy CAM Machine

Erik Ruf

January 26, 2007

**Abstract**

The Greedy CAM architecture describes a class of experimental processors that aim to cope with memory latency and enable parallelism by combining a streams-and-kernels execution model with a relational-query-based memory model. This article focuses on the programming abstraction (equivalent to the ISA[1] in more conventional systems) of Greedy CAM systems, as exemplified by a low-level intermediate language. Using a series of small example programs, we demonstrate several programming idioms and analyze their performance using a simple functional-level simulator. We also suggest extensions needed for the implementation of higher-level programming abstractions.

## 1 Introduction

The Greedy CAM architecture [Bit07] aims to improve computational performance by combining an associative, query-based streaming memory system with one or more pipelined computing engines. Such systems allow the programmer to exploit a spectrum of performance techniques, including instruction-level parallelism and pipelining, streaming memory access with data-dependent patterns, and interprocessor parallelism with shared memory- and signal-based communication and synchronization. While our eventual goal is to hide much of this complexity via automatic compilation techniques, both history and prudence suggest delaying such an effort until the performance and utility of the various architectural features and their implementations are better understood.

This article presents our experience with a single prototype instance of the Greedy CAM model, which has not yet been implemented either in real hardware or as a detailed hardware-level simulation. We have instead explored the prototype's capabilities and shortcomings at a more abstract level by surfacing its major architectural features in a low-level intermediate language, LINT. The language serves as a useful notation for exploring the design of algorithms for the architecture. Paired with an interpreter, it enables empirical verification (debugging) of algorithm implementations. Augmenting the interpreter with instrumentation yields coarse measures of eventual algorithm performance.

We begin with a summary of the general Greedy CAM model. Moving on to the prototype instance, we introduce its capabilities by presenting the syntax and informal semantics of LINT, along with a simple running example. After introducing the interpreter and its execution model, we survey a variety of programming idioms as exemplified by LINT programs, which leads to a discussion of issues and desiderata for the next prototype. The article proper concludes with a brief description of recent related work on stream-and-kernel programming models. Listings, explanations, and performance discussions of all examples are available in the Appendix.

## 2 Programming model

This section describes the general Greedy CAM programming model, and the present prototype instance as exemplified by the LINT intermediate language.

[1] Instruction Set Architecture.

```
<program> :: =
 (program
    (typedefs <typedef>*)      ;; type definitions
    (tables <table>*)          ;; tables and initialization
    (kernels <kernel>*)          ;; kernels
    (schedule <sched-expr>))   ;; inter-kernel control structures
```

Figure 1: BNF for LINT `program` construct

### 2.1 High level architecture

We briefly summarize the model described in [Bit07]:

A Greedy CAM *program* consists of an arbitrary number of *kernels* that access a shared global *memory*. The memory is organized as a set of distinct *tables*, each of which is roughly equivalent to a table in a relational database. Each table is a two-dimensional structure having a fixed number of named *columns* and a variable number of *rows*, where each row defines a mapping from column names to data values. A memory query operation is similar to a database query: it takes as arguments a list of table names, and a constraint expression that may (in the most general case) involve any column of any table being read.[2] The query operation returns a stream of tuples of rows satisfying the constraint expression.

When activated, a kernel initiates a query, and iteratively executes its body on each of the tuples contained in the query's result stream. The kernel body is a scalar expression DAG whose leaves are obtained from the query tuple, and whose roots are consumed by memory-write operations (e.g., writing a row to a table). To allow query and body execution to be pipelined, the semantics of mutating a table being queried are not defined (i.e., the query may or may not find any rows written by the body). Tuple deletion raises similar issues which we will not describe here.

While operations within a kernel are data-driven (the body DAG encodes all data dependences), the activation of kernels is control-driven using a signaling mechanism. Each kernel can raise named *signals* based on various events (e.g., an explicit *raise* operator, successful processing of an entire query stream, failure of a query, etc), and each kernel's activation is explicitly predicated on a boolean expression parameterized by signal names.

## 3 Prototype instance

This section describes a particular implementation of the general architecture described above. We will not directly describe the prototype hardware implementation (soon to be available as a low-level FPGA-based simulation). Instead, we will present the syntax and informal semantics of a low-level intermediate language, LINT, that surfaces most of the hardware's features. An interpreter for LINT programs can serve as a high-level functional simulator for any hardware implementing the LINT programming abstraction. This allows experimental evaluation of some architectural features at a far lower cost than that of redesigning the hardware implementation each time a change is made.

LINT is intended as an intermediate language to be generated by compilers and other tools, so its concrete syntax is very similar to the underlying abstract syntax tree. At the same time, we realize that a compiler effort is still off in the future, and have added some features (mostly keywords for syntactic constructs that could be deduced from position alone) to make it easier to program directly in LINT. We use an s-expression based syntax to allow for direct construction of a syntax tree without the need for complex lexing and parsing.

[2]Such a mechanism can implement a simple SQL query: constraints relating columns and constants are the equivalent of SQL "select where" specifications, while constraints relating columns are the equivalent of "select inner join" specifications.

```
<typedef> ::=
  (typedef <name> <texpr>)
<texpr> ::=
  <name> | (int <size>) | real | <struct> | <union>
<struct> ::=
  (struct (<name> <texpr>)*)  ;; may not contain reals
<union> ::=
  (union (<name> <texpr>)*)   ;; may not contain reals
```

Figure 2: BNF for LINT `typedef` construct

## 3.1 Programs

A LINT program is a declarative description of the tables, kernels, and control structures to be executed by the Greedy CAM machine. We intend that the program be completely executed by the machine, without any need for interaction with a host system. Operations to load and unload programs, or to transfer data between the Greedy CAM machine and a host, are considered to be "operating system" or "API" issues unrelated to the LINT language. A BNF description of the `program` construct is given in Figure 1. The following sections describe each of the program's subforms in greater detail.

## 3.2 Type definitions

The prototype machine supports integers of arbitrary, but fixed, length, using the standard two's complement binary encoding. It also supports a proprietary real number format based on a form of digit-serial arithmetic [Kaj07]. To simplify user-level programming, LINT provides a simple mechanism for declaring types, including integers, reals, and `C`-style structures and unions. This is shown in Figure 2.

For example, a pair of 8-bit array indices $(i, j)$ could be represented as `(int 16)` where $i$ occupies the low 8 bits, and $j$ occupies the high 8 bits. These indices could be extracted using a concrete projection operator, parameterized by an explicit bit-range constant (e.g., `7:0` for $i$ and `15:8` for $j$, respectively). Alternately, the declaration `(typedef _a (struct (i (int 8)) (j (int 8))))` [3] would enable the projection operator to be parameterized by the path constants `(path _a i)` for $i$ and `(path _a j)` for $j$ (see section 3.4.3). To enable compatibility between the two styles, structs are laid out in unpadded, big-endian order, as in `C` and `C++`.

## 3.3 Tables

While the high-level architecture supports tables with arbitrary numbers of columns, and allows all columns to participate in constraints, the prototype machine (and thus LINT) supports a more restricted form of tables and queries. All LINT tables have exactly three columns, called the *index*, *ephemeral*, and *data* columns, respectively. The first two columns are required to be integral, while the third must be scalar. Each of these columns has a specific role in the prototype's restricted query mechanism as described in Section 3.4.2.

Tables are declared and initialized using the `table` declaration, described in Figure 3. Each table has a globally unique name.[4] A table implements either a one-to-one or a one-to-many relation between index values and rows. In the former case, the `function` specifier is used, otherwise the `relation` specifier is used.

The third, fourth, and fifth arguments to a `table` declaration are the types of the index, ephemeral, and data columns, respectively. While the data column can be of any type, the index and ephemeral columns must

[3] While the type names defined by `typedef` are unrestricted, we follow a convention in which the first character of a user-declared type name is always an underscore.

[4] While future versions of the architecture may allow dynamic table naming, the prototype makes use of fixed naming to predict which regions of memory will be accessed by particular kernels.

```
<table> ::=
  (table <name> <table-kind> <index-type> <ephem-type> <data-type> <table-init>)
<table-kind> ::= function | relation
<index-type> ::= <texpr>                    ;; integral only
<ephem-type> ::= <texpr>                    ;; integral only
<data-type>  ::= <texpr>                    ;;
<table-init> ::= (<index-type> <ephem-type> <data-type>
<init-expr>  ::=
  <constant> |                              ;; scalar case
  (<init-expr>*) |                          ;; struct case
  (<name> <init-expr>)                      ;; union case
```

Figure 3: BNF for LINT `table` construct

be integral, because the hardware's query mechanism does not support constraints involving non-integral data.

The fifth argument is a list of rows (3-tuples of constant literals) used to initialize the table when the program is loaded. Constant literals can be numbers, lists of literals (for initializing structs), or $\langle name, literal \rangle$ pairs (for initializing unions). In the prototype, the only way to initialize a table is by explicit literals in the program itself. Once a mechanism for transferring data between the Greedy CAM machine and the host is finalized, LINT will be updated to support this feature.

A table representing a real-valued vector with a 16-bit index could be declared as

```
(table vec function (int 16) (int 0) real ())
```

while a table representing a real-valued matrix might be declared as

```
(table mat1 function (int 32) (int 0) real ())
```

storing both the inner and outer indices in the 32-bit index column, or as

```
(table mat2 relation (int 16) (int 16) real ())
```

storing the inner and outer indices in the index and ephemeral columns, respectively. Had we wished to initialize the matrix to the 2x3 "iota" matrix, we would have used the initializer

```
((0 0 0.0) (1 0 1.0) (2 0 2.0) (65536 0 3.0) (65537 0 4.0) (65538 0 5.0))
```

in the functional example, and

```
((0 0 0.0) (0 1 1.0) (0 2 2.0) (1 0 3.0) (1 1 4.0) (1 2 5.0))
```

in the relational example. As we shall see below (Section 5), the choice of appropriate representation depends on how the table is to be used.

### 3.4 Kernels

The kernel abstraction represents the basic unit of computation in the system. Each kernel executes queries against some number of tables. The resulting values are fed into a DAG of scalar operations that may include arithmetic, logical operations, memory writes, and signaling. The remainder of this section describes the `kernel` syntactic form as shown in Figure 4.

```
<kernel> ::=
  (kernel <name>
          (wait <name>*)
          (on-entry <action>*)
          <query>
          <body>
          (on-success <action>*)
          (on-failure <action>*))
```

Figure 4: BNF for LINT `kernel` construct

### 3.4.1 Signals and Actions

Kernel execution is controlled via an explicit signaling mechanism. Using the `wait` construct, a kernel can declare zero or more named signals, all of which must be raised before the kernel is allowed to execute. Since a given signal can be sent from multiple locations in the program, the `wait` construct is a conjunctive normal form (i.e., "AND of ORs") constraint over signal names. Other forms of constraints can be implemented at the cost of additional signaling or the use of intermediate kernels as concentrators.

Once the wait condition is satisfied, the kernel is eligible for execution. While the system guarantees that such an executable kernel will be executed eventually, there are no scheduling guarantees. For example, serial execution of kernels must be enforced by having each kernel signal its successor upon completion. Kernels without mutual dependences may or may not be executed in parallel, depending on the dynamic availability of system resources.

When a kernel is activated, it first performs a series of zero or more *entry actions* (c.f., Figure 7) listed in the `on-entry` construct. Each entry action is of the form `(clear-table <name>)`, causing all rows in the designated table to be deleted. The kernel then executes its query and processes all of the resulting tuples (see below). When the query is complete (i.e., the stream of tuples returned by the query is empty), one of two conditions will hold. Either the kernel has processed at least one tuple, in which case we says that it has *succeeded*, or it has processed no tuples, in which case we say that it has *failed.* Based on this result, the actions listed in either the `on-success` or `on-failure` constructs are executed. The allowable actions include the `clear-table` action described above, along with the ability to raise a particular signal on a particular kernel via the `(signal <kernel-name> <signal-name>)` construct.

### 3.4.2 Queries

Each kernel executes a single query that may involve multiple tables. In contrast with the general architecture, which allows for arbitrary constraints relating any columns in any of the tables taking part in the query, the prototype performs its queries in a very structured fashion that makes use of the indexed nature of tables. In database terminology, each kernel executes an "indexed nested loop join."

At a high level, ignoring low-level hardware details, the join process proceeds as follows. A list of loop nest descriptors (called `bind` expressions in LINT) is traversed by a recursive algorithm.[5] Beginning with the outermost nest, the kernel fetches one row at a time from the specified table, in order of increasing index column values,[6] subject to specified constraints on the value of the row's index column. The row

[5] We describe the join algorithm as though it was a sequential recursive algorithm. In reality, the implementation is free to execute queries in a pipelined, parallel fashion, to overlap data fetching and kernel body execution, and to eliminate interpretive overhead by software and hardware compilation techniques.

[6] For many programs, the use of functional relationships between input indices fetched in queries and output indices written to tables makes the order of query results irrelevant. The forced ordering of reads based on index column values is, however, necessary in some cases. For example, it can be used as a sorting mechanism (e.g., the priority queue in Figure 33) or to avoid data races between reads and writes to the same table (e.g., tree-based reduction codes that reuse memory at multiple tree levels, such as `vec-red-tree` on page 18).

```
<read> ::=
  (read <binding>*)
<binding> ::=
  (bind <name>                          ;; alias name
        <name>                          ;; table name
        <quantifier>
        <deletion-mode>
        (<constraint>*))
<constraint> ::= (= <bitrange>          ;; range to match in target index
                    <constraint-value>) ;; value to match against
<constraint-value> ::=
  <constant> |                          ;; fixed value
  (<name>                               ;; src table name
   <src-table-col>                      ;; src column to read
   <bitrange>))                         ;; range to read from src col
<bitrange> ::=
  <explicit-bitrange> | <path-bitrange>
<explicit-bitrange> ::= <int>:<int>
<path-bitrange> ::= (path <texpr> <name>*)
<src-table-col> ::= i | e
<quantifier> ::= forall | exists | !exists
<deletion-mode> ::= preserve | delete
```

Figure 5: BNF for LINT `read` construct

data is bound to an identifier that can be accessed both in later `bind` expressions (inner loop nests) and in the kernel body. The kernel then recursively evaluates the next innermost `bind` expression. Each time the last (innermost) binding successfully reads a row, the environment mapping table names to row values is complete, allowing the kernel body to be evaluated.[7]

While at first this may appear to be merely a cartesian product of filters, this is not the case, for several reasons:

- **Parameterization:** Each nest's constraint expression is parameterized with the variable/row bindings introduced by its enclosing nest(s). This makes it possible to enforce equality between arbitrary bit positions in the nest's index column and arbitrary bit positions within the index or ephemeral columns of any table being read by enclosing nests. This is how the "nested loop join" is achieved.
- **Qualifiers:** A particular nest need not return all rows matching its constraints. Other options include (1) returning only the first match for which all enclosed nests succeed ("exists" case) or (2) succeeding only when no matching rows are found ("not exists" case). The latter case does not bind a variable; thus, enclosed nests cannot refer to the nonexistent row.
- **Deletion:** Optionally, a nest can be configured to delete its current row tuple from the table before reading a new one. This is typically only applied to the outermost nest and any (transitively) enclosed nests having the "exists" property (i.e., only delete rows that will not be requested again during this execution of the kernel).

The query mechanism described above is reified in LINT via the `read` construct, whose BNF appears in Figure 5. Each `read` consists of a list of `bind` specifications, each of which defines a loop nest in the join. Each `bind` construct has a number of arguments:

[7] This environment is, of course, compiled to register accesses or wires and buffers, depending on the hardware used to implement kernel bodies. It should be clear that no tables or indirect lookups are required. The environment corresponds to the "operand set" concept of [Bit07].

- **Alias name:** The variable name to which the resulting row tuple is to be bound. Subsequent `bind` constructs will have this name in their scope, as will the kernel's body. Having alias names allows for more informative or convenient naming within a kernel, and allows the same table to be bound multiple times, thus achieving self-joins.
- **Table name:** The name of the table to be read by the binding.
- **Quantifier:** One of `forall`, `exists`, or `!exists`.
- **Deletion mode:** `preserve` or `delete`.
- **Constraints:** A list of constraint specifications, each of which specifies a bit range in the table's index (i.e., that which is to be constrained) and either a constant or a table, column, and bit range (i.e., source of constraint value). At present, only bitwise equality constraints are expressible; future implementations may implement other comparison operations.

We will now give a few small examples of queries in LINT. First, consider the problem of matching elements of dense tagged vectors for the purpose of executing scalar operations on equivalently-tagged elements. Given two instances `vec1` and `vec2` of the vector representation used in table `vec` on page 4, we could accomplish this via

```
(read
  (bind a a forall preserve ())
  (bind b b exists preserve ((= 15:0 (a i 15:0)))))
```

The first `bind` construct specifies that the outer join loop reads all rows of `a` (the constraint list is empty). The second binding specifies, for each row $r_1$ read from `a` by the outer loop, the inner loop will attempt to read a single row $r_2$ of `b` such that all bits in $r_2$'s index column match the corresponding bits in $r_1$'s index column.[8] The rows $r_1$ and $r_2$ can be referenced in the kernel's body via the variables `a` and `b`. As an aside, this query also supports some sparse vector operations; e.g., it would find all element pairs needed for elementwise multiplication, but would not support elementwise addition (because elements whose tags appear in only one vector would never be fetched).

Another simple example involves the use of "forall" in inner loops to implement cartesian products. Suppose we wish to find all pairs of elements in two matrices $a[i, j]$ and $b[j, k]$ having identical $j$ values (this is used in some matrix multiplication algorithms for the Greedy CAM machine). If the representation of table `mat2` on page 4 is used, the appropriate query would be

```
(read
  (bind a a forall preserve ())
  (bind b b forall preserve ((= 15:0 (a e 15:0)))))
```

which constrains $b$'s index column with $a$'s ephemeral column. If the packed representation table `mat1` (page 4 is used, the query becomes

```
(read
  (bind a a forall preserve ())
  (bind b b forall preserve ((= 15:0 (a i 31:16)))))
```

in which case we must explicitly extract the appropriate bits (the $j$ index) from `a`'s index column to form the constraint on `b`.

Finally, we give an example using aliases and both constant and derived constraints. This sort of query is common in code using tiling. For example, suppose we want to unroll the elementwise vector operation example from above. Assuming that the vector lengths are multiples of two, we can write

[8] For correctly formed vectors, in which no tag appears more than once, using `forall` in `(bind b ...)` would yield the same result.

```
(read
  (bind a0 a forall preserve ((= 0:0 0)))
  (bind a1 a exists preserve ((= 15:1 (a0 i 15:1)) (= 1:0 1)))
  (bind b0 b exists preserve ((= 15:1 (a0 i 15:1)) (= 1:0 0)))
  (bind b1 b exists preserve ((= 15:1 (a0 i 15:1)) (= 1:0 1)))
```

which explicitly constrains the least-significant bits of each binding to a constant (to fetch the correct tile element) while constraining all higher bits to be equal (to force reading of one tile at a time). More complex examples of queries can be found in the examples of Section 5.

#### 3.4.3 Bodies

Kernel bodies are small programs that process query results and perform memory writes and signaling. In the Greedy CAM prototype, these are implemented using a network of reconfigurable arithmetic units whose fringes connect to query result buffers, write units, and signaling units. To enable pipelining, kernel bodies do not have access to local state, nor do they have any form of conditional execution within the network (terminal operations such as memory writes and signaling can be predicated). We envision hiding the exact details of the network, and the scheduling of its operations, from the programmer. In a spatial analogue of the register allocation and instruction scheduling performed in traditional intermediate language compiler or virtual machine implementation, a "smart" assembler will compile LINT kernel body descriptions to network/ALU configurations, hiding the actual topology from the programmer.

In LINT, we have chosen to represent kernel bodies using a LISP-like prefix expression syntax, as described in Figure 6. We describe the various expression types here.

- **Constants**: These are numeric literals.
- **Table references:** Extracts a specified column value from the query result.
- **Arithmetic and logical operations:** See `<unary-op>` and `<binary-op>` in Figure 6. The LINT parser resolves overloaded operations and operand type constraints via type propagation and checking. Logical operations treat zero as false and all other values as true.
- **Projection:** Extracts a bit range (expressed as a range constant or a path expression) from an integral value.
- **Injection:** Returns the result of injecting a numerical value into a specified bit range of a target value.
- **Writes:** Given an integral predicate, conditionally writes a tuple to a table.
- **Signals:** Given an integral predicate, conditionally raises a signal on a kernel.
- **Local binding constructs:** New lexical scopes (binding local variables to values) are introduced using the `let` and `let*` constructs, as in Scheme. The former performs all bindings in parallel, while the latter allows later bindings to reference earlier ones.[9]

Note the absence of a selection operator that use a predicate to choose between two or more input values. This is due to pipelining concerns in the prototype hardware. Although conditional execution of the consequent and alternative could be avoided by using an eager semantics, the prototype's use of a variable-length serial numeric representation could still induce stalls. Thus, all conditional behavior must be expressed via selective execution of entire kernels (via explicit signaling or signal-like queries).

For example, the elementwise vector operation shown on page 7 might have the body

```
(let ((tag (table-index a))                    ;; extract vector index
      (sum (+ (table-data a) (table-data b))))  ;; extract and add values
  (cwrite 1 c tag 0 sum))                      ;; write result
```

[9]Note that the use of `let*` does not guarantee sequential execution; it merely allows a more convenient expression of data dependences.

```
<body> ::=
  (body <bexpr>)
<bexpr> ::=
  <name> |
  <constant> |
  <bop>
<bop> ::=
  (<unary-op> <expr>) |
  (<binary-op> <expr> <expr>) |
  <tref-op> |
  <project-op> |
  <inject-op> |
  <begin-op> |
  <let-op> |
  <write-op> |
  <signal-op> |
<unary-op> ::= - | not | lnot | int->real | real->int
<binary-op> ::=
  + | - | * | / |
  shl | shr | or | and | xor | lor | land | lxor |
  < | > | <= | >= | = | !=
<tref-op> ::=
  (tref i <name>) | (table-index <name>) |
  (tref e <name>) | (table-epehemeral <name>) |
  (tref d <name>) | (table-data <name>)
<project-op> ::=
  (project <bitrange> <expr>)
<inject-op ::=
  (inject
    <expr>           ;; destination
    <bitrange>       ;; bitrange in destination
    <expr>)          ;; source
<begin-op> ::= (begin <expr>*)
<let-op> ::=
  (<letkind> (<letbinding>*) <expr>)
<letkind> ::= let | let*
<letbinding> ::= (<name> <expr>)
<write-op> ::=
  (cwrite <expr>    ;; predicate
          <name>    ;; table name
          <expr>    ;; index value
          <expr>    ;; ephemeral value
          <expr>)   ;; data value
<signal-op> ::=
  (csignal
          <expr>    ;; predicate
          <name>    ;; kernel name
          <name>)   ;; signal name
```

Figure 6: BNF for LINT `body` construct

```
<action> ::=
  (clear-table <name>) |   ;; entry or exit actions
  (signal <name> <name>)   ;; exit actions only: <kernel-name>, <signal-name>
```

Figure 7: BNF for LINT action constructs

### 3.5 Control structures

Activation of kernels is completely controlled by the signaling mechanism described above. When execution begins, all kernels whose `wait` declarations are empty are eligible for execution, while all others must wait until their `wait`s are satisfied. Execution continues until no kernels have satisfiable `wait`s.

Although LINT is a low-level intermediate language, intended as a target for compilers that will likely need to explicitly control signaling, it is also being used for design-space exploration via hand-coded LINT programs. Since signaling code is easy to get wrong and difficult to debug, LINT also provides high-level structured control flow operations, which the parser translates (or "lowers") into explicit signals and signal operations.

The `schedule` declaration of a LINT program contains a single *schedule expression*, a tree whose internal nodes are scheduling operators and whose leaves are kernel names. The full grammatical description is given in Figure 8. We begin with a series of *primitive* schedule expressions that are directly translated into signaling constructs by the LINT parser.

- `empty`: does not execute a kernel. Always fails.
- `<name>`: executes named kernel. Succeeds/fails based on kernel.
- `(not <expr>)`: executes expr. Inverts expr's success/failure.
- `(seq <kind> <expr>*`: Sequentially executes subexpressions. Based on "kind" argument, will exit after a successful subexpression, a failed subexpression, or after all subexpressions have terminated.
- `(par <kind> <expr>*)`: Executes subexpressions in parallel, followed by a barrier.[10].
- `(loop <kind> <predicate-expr> <body-expr>)`: Iteratively executes body, exiting when predicate is true. Predicate is evaluated before or after body based on "kind."
- `(if <predicate-expr> <true-expr> <false-expr>)`: If predicate subexpression succeeds, executes true-expr, otherwise false-expr.
- `(raw <entry-name> <exit-name>)`: This construct is the interface between automatically- and manually-generated signaling code. The user-managed code must have a single "entry" kernel and a single "exit" kernel. The `raw` expression returns the same success/failure status as the "exit" kernel does.

The LINT parser also supports a number of *derived* schedule operators that, although implementable in terms of the primitive operator above, are sufficiently useful that they are provided via a form of macro expansion. As shown in Figure 8, these include sequential execution (`begin`), loops (`while`, `do`, `until`), short-circuiting logical operations (`or`, `and`) and parallel logical operations (`p-or`, `p-and`).

[10]Because kernels interact through shared memory, we need to guarantee that a kernel succeeding the parallel region sees a consistent memory that won't be altered by any not-yet-terminated members of the parallel region.

```
<sched-expr> ::=
  <prim-sched-expr> |
  <derived-sched-expr>

<prim-sched-expr> ::=
  empty  |
  <name> |                    ;; kernel name
  (not <sched-expr>)
  (seq <seq-kind> <sched-expr>*) |
  (par <par-kind> <sched-expr>*) |
  (loop
    <loop-kind>
    <sched-expr>              ;; exit predicate
    <sched-expr>) |           ;; body
  (if
    <sched-expr>              ;; predicate
    <sched-expr>              ;; succ case
    <sched-expr>)             ;; fail case
  (raw
    <name>                    ;; name of start kernel
    <name>)                   ;; name of end kernel

<seq-kind> ::= exit-succ | exit-fail | exit-last
<par-kind> ::= or | and | done
<loop-kind> ::= exit-pre | exit-post

<derived-sched-expr> ::=
  (begin <sched-expr>*) |
  (while <sched-expr>*) |     ;; first arg is predicate
  (do    <sched-expr>*) |     ;; last arg predicate
  (until <sched-expr>*) |     ;; last arg is predicate
  (or    <sched-expr>*) |
  (and   <sched-expr>*) |
  (p-or  <sched-expr>*) |
  (p-and <sched-expr>*)
```

Figure 8: BNF for LINT schedule expressions

For our simple vector addition example, the schedule consists only of a single kernel, e.g. `(schedule (vec-add))`. No signaling is required or introduced. A more complex schedule from a chart-parsing program

```
(schedule
  (begin
    init-maps
    (do
      (begin
        predict
        complete)
      (and update (not check-result)))))
```

demonstrates the convenience of the high-level control primitives.

# 4 Implementation

At present, we have constructed a parser and interpreter for LINT, implemented in approximately 2500 lines of Scheme. Once a low-level hardware specification becomes available, we expect to augment the parser with an assembler phase that produces hardware-level configuration information. This will be used to drive both software- and FPGA-based simulations.

## 4.1 Parser

The LINT parser is a hand-written recursive-descent parser. It takes LINT source code represented as a Scheme s-expression (multiply nested list) and produces a typed internal representation that can be used as input to the interpreter. The s-expression based source format is particularly convenient for machine generation (e.g., by a compiler for a future high-level language) and allows easy parameterization of test programs through the use of Scheme's backquote operator.

The internal representation differs little from the input, with the following differences:

- all symbolic references (to tables, columns, types, local variables, etc) are resolved
- all type and range expressions are in canonical forms
- all kernel-body expressions are type-checked
- all overloaded operations are bound to monomorphic signatures
- all high-level scheduling operations have been implemented via signals, output actions, and (possibly) additional empty kernels.

The translation techniques that implement these representation changes are straightforward, with the exception of the high-level scheduling primitives, whose implementation is described in Appendix 8.3.

## 4.2 Interpreter

The interpreter executes the internal program representation, modeling the effects of memory, scalar, and signaling operations. Thus, it can be used as a limited functional simulator for the prototype Greedy CAM machine. The interpreter is *sound*, but not *complete* under the semantics of the hardware. That is, while the sequence of kernel executions, memory operations, and signals simulated by the interpreter is always one that *could* have been performed by the hardware, it is not necessarily *the* sequence that the hardware *would* have performed. This is the case for several reasons:

- **Memory nondeterminism:** The order in which query tuples are returned is only partially specified. While the tuple sequence must obey an ordering relation on the index column, the ephemeral and data columns are not restricted. Furthermore, the hardware semantics only guarantees that a kernel's writes and deletions are completed by the end of the kernel's execution. Thus, it is not possible to know whether kernel $x$'s query will see the result of any writes or deletions performed either by $x$ or by any other kernels executing concurrently with $x$.
- **Scheduling nondeterminism:** The signaling mechanism enforces only a partial ordering on the execution of kernels. The hardware scheduler is thus allowed to choose both the ordering of kernels and the degree of parallelism allowed, while the interpreter uses a deterministic ordering strategy and simulates parallelism using a deterministic interleaving scheme.

To aid in the understanding and debugging of test programs, we give a more precise description of the interpreter's semantics here.

```
step := 0
signals := empty
active := empty
do
  foreach kernel k not in active where signals contains k.wait
    k.succeeded-once = false
    initialize k.query
    execute k.on-entry
    active += k
  if active not empty
    foreach kernel k in active
      read one tuple t from k.query
      if read succeeds
        perform deletions specified by k.query
        execute k.body on t
        set k.succeeded-once := true
        k.successes++
      else
        active -= k
        signals -= k.wait
        k.failures++
        if k.succeeded-once
          execute k.on-success
        else
          execute k.on-failure
    step += 1
  else exit do
```

Figure 9: Interpreter pseudocode

#### 4.2.1 Memory model

The interpreter models tables using a two-level data structure: a sorted mapping from *index* column values to lists[11] ("buckets") of $\langle$*ephemeral column value*, *data column value*$\rangle$ tuples. A memory write request ensures that an appropriate bucket exists, then adds the $\langle e, d \rangle$ tuple to the head of the bucket's list. A memory read request with a given index constraint (i.e., for each bit position in the index, either a 0, 1, or don't-care value is specified) spawns a coroutine ("iterator") of the form

```
forall index values i satisfying constraint    ;; N.B. index values are sorted
  forall tuples <e, d> associated with i       ;; N.B. in list order
    yield <i, e, d>
```

Thus, if a kernel writes (or removes) a tuple to table $t$ at index $i$, this change will be seen only by kernels whose iterator(s) on $t$ are processing an index $i' < i$. We suggest that programmers not rely on this semantics; it is an overspecification of the prototype's behavior, and may not hold under alternate implementations.

We assume that any reasonable hardware implementation will attempt to take advantage of the locality inherent in the use of buckets. Thus, the interpreter maintains statistics regarding both the number of bucket lookups and the number of intra-bucket operations on tuples.

[11] Unordered containers would suffice. For the sake of debugging and reproducible execution, we wish to keep things deterministic.

#### 4.2.2 Execution model

The interpreter uses a simplified execution model that assumes each iteration of a kernel (i.e., accessing memory to assemble a query result tuple, executing scalar operations in the kernel body, and performing writes, deletions, and signaling) takes unit time. It also assumes that infinite computational resources are available; i.e., at any given time, all kernels executable at that time will indeed be executed.

The interpreter models time as a sequence of epochs we call *steps*. At each step, the interpreter updates a list of "active" kernels whose wait constraints have been satisfied, and initializes all newly active kernels. Each active kernel is then allowed to process a single query tuple. Successful kernels remain on the active list, while unsuccessful ones execute their exit actions and are removed from the active list. Interpretation continues until the active list is empty. See Figure 9 for details.

The interpreter maintains statistics on the number of steps, the number of kernels initialized, and the numbers of successes and failures. By default, the interpreter displays the initial and final contents of all tables, followed by the statistics. Optional flags allow for the display of additional information as interpretation proceeds (see Section 8.2 for details).

# 5 Examples

This section presents a number of sample algorithms implemented in LINT. While having a variety of algorithms does help to demonstrate the versatility of the Greedy CAM model, this section's purpose lies more in the direction of demonstrating various idioms, tradeoffs, and limitations of the prototype model.

## 5.1 Mapping

One of the most fundamental operations on container data structures is the *map* operator, which applies a function to the elements of one or more containers, returning a result having the same shape as the input(s). This is easily accomplished in LINT using either explicit indexing or implicit streaming. For the sake of argument, assume that tables `a`, `b` and `c` hold dense tagged vectors whose tags are stored in the index column and whose values are stored in the data column. We wish to compute the elementwise product `(c = a * b)`.

Our first example mimics the usual pointer- or index-based mapping code by iteratively incrementing an index value (induction variable), which is used to specify the positions of the vector elements to be read or written. Since LINT kernels have no local storage, an additional table `n` is needed to hold the index value. Each execution of the kernel `add` fetches the current container index and uses it to constrain the tags of the input elements and to compute the tag for the each output element. Unlike traditional mapping loops, which terminate via an explicit count check or sentinel element value, the example terminates automatically when the index is not available in either input table:

```
;; from vec-add-iterative
(program vec-add-iterative
  ...
    (kernel add
      ...
      (read
       (bind n n exists delete ())
       (bind a a exists preserve ((= 7:0 (n i 7:0))))
       (bind b b exists preserve ((= 7:0 (n i 7:0)))))
      (body
       (begin
         (cwrite 1
                 c
                 (table-index n)
                 ((+ table-data a) (table-data b)))
         (cwrite 1 n (+ (table-index n) 1) 0 0)))
      ...)
  (schedule (while add)))
```

This code requires three reads, two writes, and one kernel initialization per vector element. As in traditional CPUs, the number of induction variable accesses can be reduced via loop unrolling (e.g., reading multiple `a` and `b` values and writing multiple `c` values per iteration). Alternately, if we are willing to destroy one of the inputs, we can eliminate the explicit induction variable (and one read and one write per iteration) by reading and deleting an arbitrary $a$ element and finding the matching element of $b$:[12]

```
;; from vec-add-iterative2
(kernel add
  (read
    (bind a a exists delete ())
    (bind b b exists preserve ((= 7:0 (a i 7:0)))))
    (body
      (cwrite 1
              c
              (table-index a)
              ((+ table-data a) (table-data b))))
  (schedule (while add)))
```

The above code is still iterative in nature: the kernel reads only one $a, b$ pair before it terminates. Successful termination sends a signal that causes the kernel to be reinitialized and executed again, while failing termination causes the program to end. A streaming approach makes better use of the execution model by executing many queries in a single kernel invocation:

[12] From this point on, we will omit ellipses to save space.

```
;; from vec-add-streaming
(kernel add
  (read
   (bind a a forall preserve ())
   (bind b b exists preserve ((= 7:0 (i i 7:0)))))
   (body
      (cwrite 1
              c
              (table-index a)
              0
              ((+ table-data a) (table-data b)))))
(schedule add)
```

This version performs two reads and one write per element, and only initializes the kernel once for the entire loop. More importantly, the input reads, addition operation, and output writes can be pipelined. Thus, streaming is clearly the preferred implementation for mapping operations in the Greedy CAM model. In streamed mapping code, unlike in the iterative algorithms, loop unrolling adds no value, as the query processing would becomes more complex, while the number of reads and writes remains unchanged.

Nested mapping is also possible, due to the "nested loop join" query mechanism. While the vector example above used the `forall` qualifier on the outermost binding only, it is also possible to use it in subsequent bindings. For example, a vector cartesian product can be generated via a doubly-nested query structure

```
(read
  (bind a a forall preserve ())
  (bind b b forall preserve ()))
```

Constraining the inner binding based on part of the result from the outer one yields results similar to the expansion phase of a two-phase expand/reduce matrix multiplication algorithm (i.e., we can match $a[i, j]$ with $b[j, k]$ for all appropriate values of $i$, $j$, and $k$.

Parallelization of both iterative and streaming codes is straightforward. Computation can be distributed across any power-of-2 number of kernels merely by constraining bit ranges within the vector indices with appropriate constants (e.g., one kernel processes even indices, the other odd indices). Using the lowest index bits for this purpose is encouraged because this keeps the query code independent of vector length; using high bits may yield slightly better storage locality but will lead to uneven workloads if the vector length is not evenly divisible by the exponent of the bit position.

The use of higher-arity or higher-rank mapping functions is also supported. One merely needs to constrain all bits of the mapped axis or axes to be equal across all inputs.[13]

## 5.2 Reduction

Another common operation on container structures is *reduce*, which reduces a structure's rank by applying a function to compose groups (usually pairs) of elements along a specified axis or axes. This can be accomplished either via iterative updating of an accumulator, or by implementing reduction trees with streaming.

The iterative approach is much like that in mapping, but with additional operations to read from and write to the accumulator table on each iteration. In our example below, we do exactly this, and also avoid maintaining an explicit index/iteration counter value by deleting the input elements as they are read.[14]

[13] A variant of the APL-style auto-reshape functionality, allowing the use of multiple inputs having distinct but "conforming" ranks or extents, can be obtained by equating only those index bits that are present in all inputs. Note that, as with much other functionality in the Greedy CAM prototype, only power-of-2 extents will be handled properly.

[14] Because we know that, in the prototype, input index columns are always read in ascending order, we can safely reduce non-commutative functions.

```
;; from vec-red-iterative
;; result in y
(table x function _tag _void real ...)
(table y function _tag _void real ((0 0 0.0))))
(kernel red-loop
  (read
    (bind x x exists delete ())
    (bind y y exists delete ()))
  (body
    ;; add element to accumulator
    (cwrite 1 y 0 0 (+ (table-data y) (table-data x)))))
(schedule (while red-loop))
```

In contrast to induction-variable-free mapping, where loop unrolling is not useful, loop unrolling is quite useful here. For example, a by-4 unrolling

```
;; from vec-red-tiled
(table x function _tag _void real ...)
(table y function _tag _void real ((0 0 0.0))))
(kernel red-loop-by4
  (read
    (bind x0 x exists delete ((= 1:0 0)))
    (bind x1 x exists delete ((= 7:2 (x0 i 7:2)) (= 1:0 1)))
    (bind x2 x exists delete ((= 7:2 (x0 i 7:2)) (= 1:0 2)))
    (bind x3 x exists delete ((= 7:2 (x0 i 7:2)) (= 1:0 3)))
    (bind y y exists delete ()))
  (body
    (let* ((sum0 (+ (table-data x0) (table-data x1)))
           (sum1 (+ (table-data x2) (table-data x3)))
           (sum2 (+ sum0 sum1)))
      ;; add partial sum to accumulator
      (cwrite 1 y 0 0 (+ (table-data y) sum2)))))
(schedule (while red-loop-by4))
```

performs the same number of input (table $x$) reads (though the dependent tiled reads $x1...x3$ may execute in parallel), but only 1/4 the number of accumulator (table $y$) reads and writes. Furthermore, the tiled kernel is initialized only 1/4 as many times, reducing scheduling and query-setup costs. Things get slightly more complicated if we wish to support input lengths that are not a multiple of the tiling factor. The necessary cleanup can be performed by executing the untiled iterative algorithm on the remaining data, or by adding an additional "fixup" kernel for each range of leftover data sizes; e.g., separately recognizing and adding the partial sum for each of the $tilesize - 1, tilesize - 2, ..., 1$ cases.

We can achieve a higher reads-to-initializations ratio by using a reduction tree strategy. Each level of the tree is evaluated in a single kernel execution; the cost is that each resulting stream of partial results must itself be reduced until a singleton result is produced:

```
;; from vec-red-tree
(table x function _tag _void real ...)
(table y function _tag _void real ((0 0 0.0))))
(kernel red-by4
  (read
    (bind x0 x forall delete ((= 1:0 0)))
    (bind x1 x exists delete ((= 7:2 (x0 i 7:2)) (= 1:0 1)))
    (bind x2 x exists delete ((= 7:2 (x0 i 7:2)) (= 1:0 2)))
    (bind x3 x exists delete ((= 7:2 (x0 i 7:2)) (= 1:0 3))))
  (body
    (let* ((sum0 (+ (table-data x0) (table-data x1)))
           (sum1 (+ (table-data x2) (table-data x3)))
           (sum2 (+ sum0 sum1)))
      ;; place partial sum in next level of tree
      (cwrite 1 x (shr (table-index x0) 2) 0 sum2)))
(schedule (while red-by4))
```

Relative to the iterative tiled version, the tree version performs a few additional reads and writes. However, many more of these memory accesses are performed in streaming mode, as the number of kernel initializations drops by a factor of four.

With respect to inputs whose size is not a power of the tile size, the tree reduction approach has more complex cleanup issues than the tiled loop strategy does. This is because "leftover" data can be introduced at each level of the reduction tree. Since all levels are stored in a single table (i.e., written back into the original vector), an iterative postpass will pick up all of the stragglers. However, for large trees at large tile sizes, it may be more efficient to perform cleanup after each level is processed (as with the tiled loop case, this can be accomplished with a small number of kernels specialized to particular sub-tile lengths). See Section 8.1.9 for an example of tree reduction cleanup code.

Reduction trees are a standard technique in data-parallel computing [HS86, Ble90]. Our streaming adaptation factors the available parallelism into two portions: a truly parallel computation within each tile, and a streaming (serial, but pipelined) outer loop across tiles. [CBZ90] used a similar approach to implement data-parallel codes on vector machines, in which vector register names correspond to positions within tiles, and vector register contents correspond to (limited-length portions of) streams.

We can add another level of parallelism to our reduction codes by splitting the computation among multiple kernels. The simplest strategy is to parallelize on axes other than the reduction axis; e.g., if reducing $k$ vectors, simply assign $k/|vectors|$ vectors to each of $k$ kernels. If that is not possible (i.e., there are no non-reduction axes), we can assign subvectors to kernels, then combine the subvector results in an additional "final composition" kernel. Note that, if the function being reduced is commutative, we can assign each kernel a stride-based slice rather than a contiguous subvector. Doing so allows us to write size-independent code.

## 5.3 Combining mapping and reduction

A variety of algorithms can be expressed as combinations of mapping and reduction phases. The canonical example is matrix multiplication, but others (such as convolution and regular expression parsing) also exist. The primary implementation tradeoff in such situations is the choice between

1. performing the mapping and reduction operations separately, and
2. merging them.

Choice (1) allows the use of optimal implementation techniques for both phases, at the cost of passing an intermediate result through memory, while choice (2) eliminates the intermediate table but may access its input and outputs in a less efficient manner.

Consider two implementations of matrix multiplication (see Section 8.1.3 for code and statistics):

- *mat-mult-separate* uses nested *forall* qualifiers to generate the complete stream of products $a[i,j]*b[j,k]$. It writes these results to memory and invokes an arity-4 tree reduction algorithm on the $j$ dimension of the intermediate data structure.

- *mat-mult-combined* repeatedly executes a single kernel on increasing values of the $j$ index. Each pass streams in one two-dimensional plane from each of the inputs $a$ and $b$, and the accumulator $c$, and performs $c[i,k]+ = a[i,j] * b[j,k]$ for the specified value of $j$. The explicit iteration over $j$ values is required to avoid intra-kernel write-read dependences on $c$.

Experiments on 16 x 16 matrices (see Figure 26) show the separate case requiring 9728 reads and 5376 writes in 4 kernel instances, with the combined case requiring only 8481 reads and 4112 writes in 33 kernel instances. The combined strategy executes 15% fewer memory reads and 20% fewer writes, but executes 850% more queries (kernel initializations) in doing so. Without a cycle-accurate simulator, we can't really predict which will be faster (a rough measure, dividing the additional reads by the additional queries, suggests that the combined version will be faster, provided that the cost of starting a kernel is less than that of performing 43 streaming reads). In any case, it is clear that the choice of optimal strategy will depend crucially on the relative costs of operations in the actual hardware executing the program—yielding the same need for specialized hardware-specific codes within a single ISA as in traditional architectures.

Interestingly, in sparse matrix multiplication (see Section 5.4), the separate approach is clearly better than the combined approach, even though an iterative reduction scheme is used in both cases.[15] This is due to the overhead of initializing the accumulator, 'which requires execution of the same query used to implement the entire first phase of the separate approach.

In matrix multiplication, the merging strategy involved transforming a streaming axis ($j$ in this case) into an iterative one, so that the two iterative loops (expansion and reduction) could be merged. In other cases, we can instead merge a mapping kernel into the first streaming pass of a tree-based reduction kernel. For example, the kernel `input-to-maps-by4` in Figure 46 performs both an input-to-function mapping and a 4-wide function composition, saving one complete pass over the input stream.

## 5.4 Sparse data structures

Thus far, we have dealt exclusively with *dense* data structures, in which a data structure's index (and possibly ephemeral) columns have a straightforward, implicit, and usually affine mapping to and from contiguous linear memory addresses. The utility of the explicitly "tagged" data structures in LINT programs is perhaps more obvious in the *sparse* case.

We first look at sparse vectors. These have the same table structure as dense vectors, but all rows having data value 0 are elided. This representation has repercussions for both mapping and reduction applications.

In the mapping case, the appropriate strategy varies based on whether or not a missing (default) input implies a missing output.[16] For example, the streaming code[17] for mapping multiplication over pairs of vectors remains unchanged from the dense version. Values with matching indices will be multiplied and written to the result, while all non-matching cases represent multiplication by zero, whose result is not written to the output. The elementwise multiplications in sparse matrix codes can be treated in a similar manner. However, if we switch the mapped function from `*` to `+`, additional code is required. Not only must pairs of rows with matching indices be handled (i.e., added together) , but rows not matching other rows must also be handled (i.e., the non-default value must be propagated to the output). The example code below achieves this by executing three kernels in parallel: one that adds matching pairs, one that propagates

[15] The tree-based reduction strategy can't be applied to sparse data.

[16] Another way of saying this is that the operator is *strict* in the sparse data structure's default element value.

[17] Iterative codes that rely on an explicit loop-counter induction variable (e.g., the `vec-add-iterative` example on page 15) will not work, as the enclosing loop will exit when the first non-matching index is found. Iterative codes based on the use of iteration over `exists` qualifiers (e.g., `vec-add-iterative2` on page 5.1) will work properly.

singletons from `a`, and one that propagates singletons from `b`:

```
;; from sparse-vec-add-par
(kernels
  (kernel add-pairs
    (read
      (bind a a forall preserve ())                      ;; read all rows of a
      (bind b b exists preserve ((= 7:0 (a i 7:0)))))    ;; index b with a's index only
    (body
      (let ((result (+ (table-data a) (tref b d))))
            (cwrite 1 c (table-index a) 0 result))))
  (kernel add-a-only
    (read
      (bind a a forall  preserve ())                     ;; read all rows of a
      (bind b b !exists preserve ((= 7:0 (a i 7:0)))))   ;; don't allow a match on b
    (body
      (let ((result (table-data a)))
        (cwrite 1 c (table-index a) 0 result))))
  (kernel add-b-only
    (read
      (bind b b forall  preserve ())                     ;; read all rows of b
      (bind a a !exists preserve ((= 7:0 (b i 7:0)))))   ;; don't allow a match on a
    (body
      (let ((result (table-data b)))
        (cwrite 1 c (table-index b) 0 result)))))
(schedule (par done add-pairs add-a-only add-b-only))
```

Note the use of the `!exists` qualifier to establish the negation of a constraint. Depending on the hardware implementation, this might be an inexpensive lookup, or an expensive search. Should this be a problem, an alternative two-phase algorithm can be used. The first phase performs addition on all matching pairs, deleting the corresponding input rows. After this is complete, two parallel kernels copy the remaining rows from the two input tables to the result.

Sparse data structures are not generally amenable to mapping based on loop unrolling or tiled streaming, as the chance of finding sufficient contiguous indices to fill a tile is small, and the corresponding non-tiled cleanup is expensive.

Iterative sparse reduction codes have issues similar to mapping codes: operators that are strict w.r.t. the default value are simple, while others may require additional processing. Tiled codes are again impractical. In some cases, it might be most practical to compress a sparse reduction axis to contiguous indices, allowing for loop unrolling, streaming, and parallelism as in the dense case. However, until the hardware provides either a built-in compression operator or kernel-local state that can be updated during streaming, the compression algorithm alone will have the same performance as iterative reduction, due to the need to update the dense output index counter upon each write.

It is worth noting that the restriction to non-unrolled iterative codes applies only to the axes being reduced. For example, consider a sparse matrix multiplication algorithm having two phases of one kernel each:

```
;; from sparse-mat-mult-expanding
(kernel expand
  (read
    (bind a a forall preserve ())                                ;; read all rows of a
    (bind b b forall preserve ((= (path _j) (a e (path _j)))))) ;; join a,b on j coord
  (body
    (let ((product (* (table-data a) (table-data b)))
          (i (table-index a))
          (j (table-ephemeral a))
          (k (table-ephemeral b)))
      (let ((iktag (inject i (path _ik k) k)))     ;; compute output tag (concat k i)
        (cwrite 1 e j iktag product)               ;; write project to ijk space
        (cwrite 1 c iktag  0 0.0)                  ;; initialize accumulator
        (cwrite 1 js j 0 0)))))                    ;; remember that this "j" plane contains data
(kernel reduce-add
  (read
    (bind js js exists delete ())                                 ;; index of j plane in ijk space
    (bind e e forall preserve ((= (path _j) (js i (path _j))))))  ;; elts of j plane
    (bind c c forall preserve ((= (path _ik) (e e (path _ik))))) ;; else of accum plane
  (body
    (let ((sum (+ (tref e d) (tref c d))))
      (cwrite 1 c (tref c i) 0 sum))))              ;; write sum to accum, at read's index value
(schedule (begin expand (while reduce-add)))
```

The initial phase computes all matching products, writing them to a table indexed by $i$, $j$, and $k$ coordinates. The second phase iterates over all relevant $j$ coordinates, but is still able to stream over all $i$ and $k$ values relevant to a particular value for $j$.

Parallelization of sparse codes is difficult. While we can easily partition computations across a dense axis, the various per-partition kernels may have vastly different execution times due to the sparsity of the remaining axes. Partitioning along a sparse axis is just as bad: the regular (stride-based) partitioning supported by the hardware (via bit-level constant constraints) cannot adapt to irregular distributions of index column values.

In many systems, such cases are dealt with through the use of dynamic task assignment techniques such as work lists or queues (i.e., containers with atomic insertion and removal properties). Unfortunately, due to memory pipelining issues, the prototype Greedy CAM architecture's memory model does not provide atomic read-and-delete operations across multiple simultaneously-executing kernels. Thus, we are forced to compress the sparse index into a dense one (e.g., dynamically augmenting the sparse indices with contiguous "sequence number" tags), allowing the use of standard partitioning strategies. As we saw with tiling, a complete iterative traversal of the data structure is required.

## 5.5 Sorting

The fact that the query mechanism always traverses tables in ascending-index-column order is useful in several ways. Most obviously, it can be used to implement integer sorting by simply writing the sort keys into the index columns of a table, then querying the table with the index column left unconstrained. Another use was demonstrated in the `vec-red-iterative2` example, where the deterministic ordering allowed the reduction of non-commutative functions without an explicit induction variable.

The explicit index ordering property is also useful for implementing priority queues. We can structure the queue as a relational table where the priority value (0 being highest) is carried in the index column, and any additional information is carried in the ephemeral and/or data columns. This allows for multiple entries with equivalent priorities (since multiple rows may have identical index values), but does not guarantee fairness, as the memory model does not specify the ordering among rows having equal index column values. It is also possible to carry additional non-priority information in the index column, provided that it is placed in a

digit range that is less significant than that holding the priority values. For an example of this, see the `mst` example in the Appendix.

## 5.6 Negation

The `!exists` query qualifier allows a weak form of negation to be performed. We saw one example of this above, in the parallel implementation of a sparse vector addition, where it was used to distinguish the elements existing in only one of the two input vectors. It is also useful in parsing and recognition codes for establishing that all input has been consumed, and in memoizing codes where it can be used to avoid redundant computation.

It is worth noting that the same effect can often be obtained without the use of `!exists`. For example, in the sparse vector addition case, deleting all inputs with matching index pairs allows direct queries for singleton inputs, but at the cost of (1) consuming the input, and (2) reduced parallelism. Consumption of input can be verified by constructing a kernel that attempts to read the input, and using the kernel's success or failure to trigger control flow, e.g., `(schedule (until <do-something> (not <attempt-read>)))`. Similarly, memoization can sometimes be achieved via delay; i.e., writing all task descriptions to a `function`-qualified table, then traversing the table. All of these solutions have the disadvantage that they operate at kernel-level granularity, rather than query-level granularity, which means (among other things) that streaming is interrupted, possibly affecting performance.

So why even consider these solutions? The issue here is that, in higher-level Greedy CAM implementations, negation may become far more expensive. The reason `!exists` is efficient in the current prototype is that all query operations examine only indices, whose presence or absence can be determined in constant time. Should the architecture move to a full database model, a negated constraint might require searching an entire table. Also, the use of deletion to implement some uses of `!exists` will become more palatable if and when transactional deletion is supported.

## 5.7 Tiling

Tiling, also known as blocking, is a well-known implementation technique in which one or more axes of a loop's iteration space are each split into an outer loop with a large stride which provides the initial value for an inner loop having a small stride. The inner loop is then inlined, exposing parallelism and eliminating intermediate memory accesses.

In the Greedy CAM world, tiling amounts to replicating one or more of a kernel's `bind` expressions, such that the initial and duplicated `bind` forms have differing constant constraints on some (usually low-order) shared index bitrange. A variety of benefits can be achieved. In Section 5.2, we saw the benefits of tiling for both iterative and tree-based reduction codes.

Tiling is most useful when the algorithm performs a reduction-like operation on its input or some intermediate value, because it is the composition of multiple instances of the elementwise combining function that enables the substitution of intra-kernel-body variable bindings for memory operations. When applied to iterative reduction schemes, tiling amounts to unrolling the reduction loop, while in tree-based schemes, tiling corresponds to an increase in the arity of the tree nodes. A variety of algorithms, including matrix multiplication, state machine evaluation, prefix scan computation, and convolution, can be implemented on the Greedy CAM prototype using tiling techniques (see Section 8.1).

In simple cases such as vector reduction, tiling serves mainly to eliminate accumulator accesses. In more complex codes, it also enables reuse of data values that would otherwise have to be loaded multiple times. Both matrix multiplication and convolution benefit from such reuse.

In multi-pass algorithms, tiling can induce additional data representation issues. In the case of two-pass matrix multiplication, the expansion phase is most efficient when tiled along both dimensions of both input arrays, but doing so limits the efficiency of the subsequent reduction phase. Tiling along only the shared dimension achieves fewer arithmetic operations per tile in the expansion kernel, but halves the number of values the must be combined by the reduction kernel. For details, see Section 8.1.3.

Finally, we make a distinction between *logical* tiling, in which related values are fetched at one time to exploit reuse and reduce memory traffic, and *physical* tiling, in which adjacent elements are fetched at once to take advantage of spatial locality in the memory system. In the high-level Greedy CAM model, there is no concept of spatial locality at the index level; e.g., fetching multiple rows having logically related indices (e.g., $k$ consecutive elements of a vector) does not necessarily improve the execution speed of the query.[18] If the tile elements are sufficiently small integers, we can achieve physical tiling and spatial locality by packing such elements into the index and/or ephemeral columns. This is not possible for reals, as the present memory abstraction restricts us to a single data column. A possible alternative would be to pack multiple tile elements into successive identically-indexed rows with distinct ephemeral column values, but the prototype lacks a means for constraining ephemeral columns, making tile assembly impossible. Sections 6.2 and 6.6 discuss this matters in more detail.

### 5.8 Iteration over streaming

Another common idiom in the Greedy CAM model is the use of iteration to drive multiple activations of a streaming kernel. We see this both in explicit forms (e.g., the "combined" matrix multiplication algorithms iterate over consecutive $j$ indices, executing a streaming query for the handling of each plane), and implicit ones (e.g., tree reduction terminates when only a single element remains in the buffer). Sometimes, both methods are used: for example, the streaming implementation of prefix scan implicitly processes a tree in a reduction-like manner until only a single tile remains, then walks back down the tree, propagating values downward until the input level is reached. Streaming recognition algorithms iteratively combine state transfer functions (or partial parses) until the input is fully covered.

In all of these cases, what we are really seeing is a stratification of data dependences, as induced by the memory model, which does not permit a kernel activation to see the results of its own writes. This is both a blessing and a curse, in that it enables efficient pipelined memory access, but limits such processing to inner loops. In such algorithms, tiling becomes very important, because it allows for local reuse of previously computed values that would simply be loop-carried temporaries in a conventional system. What we cannot easily achieve are sliding-window optimizations such as software pipelining [Lam88], unless we restrict them to operating within tiles, as we did in convolution.[19] On the other hand, parallelization becomes fairly simple, since any streaming pass can be partitioned among any power-of-2 number of processors. Whether the performance benefits of streaming memory access and kernel-level parallelism will outweigh their overhead remains to be seen.

## 6 Discussion

This section describes various programming issues that were discovered during the development of the interpreter and example code.

### 6.1 Constraint sources

In the present prototype, binding constraints are used to specify the exact values of particular bit ranges in the index column of the table being queried. The left side of each constraint pair $\langle tgtRange, srcValue \rangle$ specifies the bit range to be constrained, while the right side specifies the value that the bit range must contain: either a constant or a reference to a bitrange in either the index or ephemeral column of a row bound by an earlier `bind` construct.

[18] In the prototype, which uses a trie-based structure to implement index lookup, some amount of locality is achieved because adjacent index values may share a common trie node.

[19] The needed bridge between tiles might be provided by a small amount of local storage (see Section 6.7), at the cost of having to process the tiles in a specific sequential order.

An earlier version of the Greedy CAM architecture had allowed the constraint value to be extracted only from the immediately preceding `bind` form, rather than any preceding one. In some cases, this caused no problems; for example, the by-2 tiled binding example

```
(read
  (bind a-0 a forall preserve ((= 0:0 0)))
  (bind a-1 a exists preserve ((= 15:1 (a-0 i 15:1)) (= 1:0 1)))
  (bind b-0 b exists preserve ((= 15:1 (a-0 i 15:1)) (= 1:0 0)))
  (bind b-1 b exists preserve ((= 15:1 (a-0 i 15:1)) (= 1:0 1)))
```

was expressible as

```
(read
  (bind a-0 a forall preserve ((= 0:0 0)))
  (bind a-1 a exists preserve ((= 15:1 (a-0 i 15:1)) (= 1:0 1)))
  (bind b-0 b exists preserve ((= 15:1 (a-1 i 15:1)) (= 1:0 0)))
  (bind b-1 b exists preserve ((= 15:1 (b-0 i 15:1)) (= 1:0 1)))
```

Other examples, particularly those involving multi-dimensional lookup tables, were more problematic. Consider the state transition function of a deterministic finite state machine:

$$trans : state \times input \rightarrow state'$$

which can be implemented as a table

```
(table vec function ((struct (s _state)) (i _input)) _state _void)
```

A kernel wishing to compute the new state given the current state and input value would like to execute a simple lookup

```
;; fetch input
(bind i i exists delete ())
;; fetch state
(bind s s exists delete ())
;; perform table lookup
(bind ns trans-table exists preserve ((= (path _si s) (s i (path _state)))
                                      (= (path _si i) (i e (path _input)))))
```

which clearly needs to refer to both the state and input values. Without this multiple-source property, we would instead have to write

```
;; fetch input
(bind i i exists delete ())
;; find all transitions accepting input
(bind t trans-table exists preserve ((= (path _transition i) (i e (path _input)))))
;; given a transition, see if it exits the current state
(bind s s exists delete ((= (path _state) (t i (path _transition src))))))
```

which is $O(|states|)$ more expensive because we have to enumerate all candidate transitions.[20] Other cases, such as parsing, can require the construction of an entire intermediate table.

This has been fixed in the current prototype.

[20] Alternately, we could bind the current state, find all transitions exiting the state, and find one consuming the current input. Whether this version is better or worse than the example code depends on which is larger: the input alphabet or the set of states. In either case, irrelevant transitions will be read and discarded.

## 6.2 Constraint targets

As described in Section 4.2.1, the prototype Greedy CAM machine represents tables using an associative memory whose keys are index column values and whose values are buckets (unsorted containers) of ⟨*ephemeral column value*, *data column value*⟩ tuples. Queries are allowed to constrain only the keys (indices). While (the ephemeral portion of) bucket tuples can contribute to subsequent query constraints, they cannot themselves be constrained.

Why does this matter? First, we note that this restriction does not limit the semantic power of the abstraction—assuming that the hardware support sufficiently wide indices, any information in an ephemeral column which we might wish to constrain can always be placed in the corresponding index column instead. The main issue at hand is memory performance; i.e., the effective use of DRAM page technology. Following the index data structure to find an index column value matching the specified constraints may require several non-contiguous memory references, but once a bucket is found, the contiguously-stored bucket tuples should be accessible in a faster, memory-page-local addressing mode. This suggests that LINT code be written to maximize the number of tuples read per unique index column value.

In some cases, this can be accomplished in the LINT code alone. Multidimensional lookup tables can easily be encoded as `relation` tables with all but the innermost axis stored in the index column, while the innermost axis is stored in the ephemeral column. When the table appears in a `forall` query, the innermost axis is enumerated using contiguous memory accesses. This can be seen in the expansion phase of the two-phase matrix multiplication algorithms `mat-mult-separate` and `sparse-mat-mult-separate`, and in the node-successor enumeration query in the kernel `process-succs` of `mst`. Examining the statistics for these kernels shows fewer bucket initializations than bucket reads, indicating that bucket contents are being reused.

Unfortunately, we often need to constrain bit positions within the inner index. For example, in the single-pass matrix multiplication algorithm `mat-mul-combined`, the $i$ and $k$ indices of the accumulator must match the $i$ index of $a$ and the $k$ index of $b$, requiring that both the inner and outer dimensions of the accumulator be represented in the index column. We could still take advantage of the bucket locality in the input matrices, at the expense of maintaining three matrix layouts: indexed outer dimension (in $a$), indexed inner dimension (in $b$) and both dimensions indexed (in $c$). Using non-uniform representations of this sort could require either explicit data transformation (akin to transposition) for cross-kernel "impedance matching", or else specialization of kernels producing or consuming the input and output matrices.

In tiling codes, all tiled dimensions must be represented in the index column because all bits describing such dimensions are constrained (either to match another tile, or to have a specific constant intra-tile position). If all dimensions are tiled, all dimension indices must be contained in the index column, and specialization is not possible.

Thus, when perusing the examples, one sees very little use of `relation` tables and ephemeral columns. We could make better use of the bucket abstraction if we were able to constrain bit ranges within the ephemeral column. Doing so would complicate the hardware somewhat, due to the need to filter bucket tuple reads. It would also allow the user to, perhaps unconsciously, replace constant-time lookup (using the index column data structure) with linear-time search (over bucket contents). Yet, this is exactly the "to index, or not to index" decision that database programmers have been making for decades. Allowing constraints on both index and ephemeral columns would give Greedy CAM programmers this same flexibility.[21]

Of course, deciding when to use which style of lookup can be difficult, even without the additional possibility of ephemeral-column constraints. Consider the problem of composing simple integer-to-integer functions represented as unordered lists of ⟨*domain*, *codomain*⟩ pairs.[22] One straightforward approach would place domain elements in the index column, and codomain elements in either the index or ephemeral columns

[21] While we have experimented with fully-relational and fully-indexed Greedy CAM prototypes, we do not as yet have a mixed implementation. Once the relative costs of bucket lookup, bucket-local read, filtering, etc. are known, a more precise mixed implementation would appear to be a fruitful direction for future experimentation.

[22] Such composition operations form the core of a number of parallel algorithms; for example, see the parallel FSM example `fsm-tree-by4-figure1` in the Appendix.

(functions are many-to-one, so we won't ever have more than one row per index value). Composing functions becomes a simple join operation:

```
(read
  (bind f f forall preserve ())
  (bind g g exists preserve (= (path _dom) (e (path _dom)))))
(body
  (write h (table-index f) (table-ephemeral g) _void))
```

For each domain/codomain mapping in $f$[23], we find all mappings in $g$ such that $f$'s output value matches $g$'s input value, and write the derived $g \circ f$ mappings into $h$. Note that $h$ must be declared as `function` so that duplicate mappings will be quashed.

This works fine, but since the query on $g$ returns at most a single result, no bucket reuse is possible. We can approach the problem differently by noticing that reversing the domain/codomain mapping yields a relation, and that these relations can be composed using a slight variation on function composition. Let each mapping now be implemented with the codomain values in the index column and the domain values (of which there may be more than one per codomain value) in the ephemeral column. These relations can be composed using

```
(read
  (bind g' g' forall preserve ())
  (bind f' f' forall preserve (= (path _dom) (e (path _dom)))))
(body
  (write h' (table-index g') (table-ephemeral f') _void))
```

where all three tables are declared with `relation`. After the code is complete, the resulting mapping $h'$ can be interpreted as a function either by using ephemeral-column constraints (if available) in the consuming code, or by executing a separate kernel to invert the mapping. Whether this is faster than the functional approach depends strongly upon the functions being composed. In all cases, there is a high risk of duplicate effort because there is no way to suppress the writing of duplicate edges into $h'$.[24] This suggests that the composition-of-inverted-mappings strategy might be practical only if the hardware were to support duplicate elimination on entire rows, rather than merely on index column values (as is currently provided for `function` tables).

## 6.3 Deletion

The `delete` query mode serves several roles in LINT programs. Perhaps the most obvious use is for storage reclamation: by deleting input rows, a kernel can free memory. This is less useful than one might think, given that entire tables can be cleared more efficiently upon kernel termination (or successor initiation) using `clear-table` actions. Depending on the degree to which hardware memory is virtualized and the sophistication of the memory allocation strategy, deletion of individual rows may or may not have a significant effect on memory usage and/or performance.

Semantically speaking, deletion does have several important uses, all of which apply to iterative computations; i.e., those that repeatedly invoke one or more kernels, which may themselves perform streaming operations.

- **establishment of completion:** While streaming operations, which occur strictly within kernels, terminate automatically when query processing is complete, inter-kernel iterative algorithms must

[23] The use of separate tables for $f$ and $g$ is merely for exposition. In a reduction scenario (e.g., `fsm-tree-by4`, shown in (Figs. 46–50), both `bind` clauses may refer to distinct index ranges within a single table.

[24] The duplicate effort mentioned here lies not in the discovery of an edge that happens to match an existing one—there's no way to know if a new edge is or is not redundant until it is found. The issue here lies in avoiding creating duplicate work for some downstream composition (e.g., the next composition in a reduction sequence) by failing to canonicalize the output data.

decide when to terminate. A simple, convenient way to do this is to have the loop body consume the input by deleting it. Once no input is available, the kernel will fail, and the failure action can be used to terminate the iteration.

- **replacement:** Kernels often maintain cross-iteration state in fixed-size tables that are updated on each kernel execution. Examples include induction variables and summary values (e.g., "current minimum"). Often, these values are maintained in index columns so that they may participate in constraint processing. If this is the case, it is necessary to remove the previous iteration's information (one or more table rows) via deletion so that the new state is not contaminated by any old rows left behind. Note that, in cases where the information need not be maintained in the index column, a fixed index value and a `function` declaration can ensure that a write will replace the row written by the previous iteration.

- **avoidance of reprocessing:** Kernels are often repetitively executed on a single table or group of tables until a termination condition is met. Often, these tables vary in size (e.g., queues, or tables representing the current level in a reduction tree). Each time the kernel runs, it must remove the input it consumes (lest an element never be dequeued, or an intermediate result in a reduction tree be added into the result multiple times). In some cases, a similar effect can be achieved using negation; i.e., writing the "used" indices to a table, and using `!exists` to avoid matching such cases—though deletion is likely to be more efficient, as it does not require search.

In spite of its utility, the present deletion semantics raises some problems. The first, shared with writing, is that the interleaving of write and delete operations emanating from multiple kernels executing in parallel is undefined. Thus, as we have noted before in comments on parallelization, concurrent shared use of tables (both as fixed inputs and as dynamically-changing queues) must be statically partitioned. Solving this problem will likely require additional locking or transaction mechanisms that are beyond the scope of this article.

More surprisingly, deletion also raises issues within the execution of a single kernel, in which the semantics are more strictly defined. Each time a `bind` form with a `delete` qualifier is stepped, the hardware guarantees that, should the currently bound row become part of a query result tuple (i.e., all subsequent binding constructs succeeded at least once), the row will be deleted from memory before the tuple becomes available to the kernel body.[25] This raises the issue of incomplete loop nests: A query containing a deleting binding construct $b$ (transitively) preceded by any binding construct $b'$ whose qualifier is `forall` may return a smaller set of query tuples than an otherwise identical query in which $b'$ has the `preserves` qualifier. Because deletions are performed eagerly during query execution, later invocations of the inner query $b$ will see the result of $b$'s earlier deletions. We recommend that deletion be performed outside of all `forall` queries. As a consequence, streaming mapping and reduction codes (implemented using `forall`) cannot easily handle mapping/reduction functions that perform deletion. Sometimes, this can be addressed by reordering the loop nests (both those explicit in the kernel scheduling and those implicit in the query binding order) such that deletion occurs only in the outermost loops.

A more transactional form of deletion, in which all of a kernel execution's deletions are queued and delayed until the kernel terminates, would solve the above problems, but would itself interfere with memory writes performed by the kernel body—meaning that writes would also have to be delayed, so that they execute only after all of the delayed deletions have been performed. We leave this to future work.

## 6.4 Affine counters

Another possible extension to the Greedy CAM model involves counting. In the current prototype, kernels have no local state whatsoever, thus requiring that any induction variables either (1) be present in the input stream (e.g., the index column of a densely tagged vector) or (2) be explicitly represented in memory. The

[25] Thus guaranteeing that the deletion will precede any writes performed by the body invocation associated with the deletion's query result tuple.

former is not always possible (e.g., the elements of a sparse vector are not sequentially tagged), while the latter compromises performance by requiring reinitialization of the kernel each time the counter's memory location is updated.

One way to address this problem would be to allow "virtual" bindings whose values are defined by scalar affine expressions (i.e., integer functions parameterized by a static base, count, and stride). These would be used in queries just like `bind` forms, but the $n$th read from these bindings would return an artificially-constructed row with index value $i = b * n + s$. The $n + 1$st query and all subsequent queries would fail.

This mechanism would be useful both for generating index values, and for use in restricting subsequent queries. Generation is useful for tasks like compressing sparse vectors, in which the sequential index values are written into the index column of the target vector. Another, similar use is the initial tagging of dense value streams provided by a host processor. Restriction provides a means for extracting subspaces of the input whose dimension sizes are not powers of 2. Restriction can also be used, if somewhat clumsily, for tiling; a 3-tiling of a vector could be specified via counters $(3n)$, $(3n + 1)$, and $(3n + 3)$. However, efficiency is diminished by the use of a counter by tile element, and by the fact that the use of counters forces an ordering on the tiles (ideally, we do not want to constrain the order in which the tiles are fetched).

## 6.5 Arithmetic in constraints

Another useful extension would allow limited arithmetic on the outputs of projection operations in `bind` expressions. The main purpose of this proposal is to allow small offsets; e.g., to constrain an index bitrange to be some previous binding's bitrange plus an integer, as in

```
(bind a0 a forall preserve ())
(bind a1 a exists preserve (= i (+ 1 (a0 i 7:0))))
(bind a2 a exists preserve (= i (+ 2 (a0 i 7:0))))
```
[26]

Such functionality would make it easier to write sliding-window codes[27], and would reduce the number of counters needed in non-power-of-2 tiling situations from the tile size to 1 (per non-power-of-2 dimension). We expect that, in hardware terms, this would be significantly more expensive than the present bitwise-only constraint scheme.

## 6.6 Additional data columns

At the moment, each table row is restricted to having only a single (integral or real) data column. While multiple integral values can be packed into the index and ephemeral columns, only a single real value can be stored per row. Thus, all data structures involving multiple real values (or more integral bits than will fit in the integral columns) must be expressed as multiple tables, to be joined whenever they are used. Examples include pairs (complex numbers, ranges), general tuples, and tiles. We could reduce both memory usage and matching overhead significantly by allowing for multiple data columns in tables.[28] One way to provide this without the need for per-row overhead is to parameterize column counts at the table level, as in databases.

## 6.7 Local state

Many operations, particularly reduction, could be simplified through the use of local state. In the prototype, a kernel body has access only to the values bound by its query, and is thus locally "functional" in nature. Any form of accumulator must be implemented via writes to memory which cannot be read without a

[26] The expression `(bind a2 a exists preserve (= i (+ 1 (a1 i 7:0))))` would also work, but might create delays since it cannot be evaluated until after `a1` has been bound.

[27] Unless some form of caching is used, this is likely to be less efficient than solutions with explicit reuse, e.g., `convolve-tiled`, because each element will be read multiple times.

[28] Allowing multiple data elements encourages the use of physical, rather than logical, tiles. The former require less matching and improve memory locality.

reinitialization of the kernel, thus requiring the use of multiple-pass techniques (and additional memory accesses). However, adding local state to the Greedy CAM model brings up a number of issues:

- **Initialization:** Local storage must be initialized at kernel instantiation time. Provided that we are willing to initialize using constant values, this should be a simple extension to the kernel description. A more ambitious extension might allow a table to be loaded into local storage (perhaps as part of the kernel entry action mechanism), enabling a more dynamic choice of initial values.
- **Finalization:** When a kernel terminates, it may require a mechanism for exporting some or all of its local state into global state (e.g., , a local-accumulator-based reduction scheme needs to propagate its result to other kernels). This suggests that we add some form of copy operation to the kernel exit action mechanism.
- **Size:** Like a register file in a traditional architecture, kernel-local storage will necessarily be of rather limited size. Ideally, the storage would be of sufficient size to represent an entire accumulator (e.g., the result pane in a matrix multiplication kernel), but this may not be possible for large data structures, in which case tiling will be necessary.
- **Indexed access:** Given that local storage will need to be related to query results (e.g., the index of a data value will determine which element of the accumulator will be updated), the local storage must be dynamically indexed. Thus, a set of named registers will not suffice. At the same time, adding a completely new indexing and updating mechanism for local storage would greatly complicate the programming abstraction (not to mention the hardware). For the sake of consistency, we could instead add a new kind of table that (1) is local to a particular kernel, and (2) guarantees that all read and write dependences (including those carried by the implicit query-result-processing loop) be honored. Doing things this way would enable re-use of the existing query syntax, and is consistent with the use of other specialized table kinds, such as the "counter" tables described above.
- **Dependences:** The major issue with local storage is that it is only useful if it preserves dependences. For example, if the first query result causes the body to write a value into a local table, a read of the same table during the processing of the second query result should return the value written by the earlier write. This sort of feedback dependence, which in the most general case may span an arbitrary number of query result processing instances, conflicts with the elementwise, dependence-free processing requirements of pipelining. Thus, to enable the use of local storage, we would have to introduce the notion of a storage-access-related pipeline stall, which might significantly slow kernel body execution (though this is still likely to be faster than the (terminate kernel, send signal, receive signal, initialize kernel) sequence required for the reuse of global state). Perhaps more troubling is the additional complexity added to the hardware that processes kernel bodies, which can no longer be implemented as a simple, unidirectional arithmetic mesh. Indeed, if the use of local storage is unavoidable, we might choose to use conventional processing technology to implement kernel bodies.

## 6.8 Higher-level language issues

We now consider the suitability of LINT as a target language for the compilation of higher-level abstractions of the Greedy CAM model. The term "higher-level" has many connotations; we will consider two here: resource virtualization and fully-relational queries.

### 6.8.1 Virtualization

Most computer languages that support operations beyond the "assembly code" level attempt to shield the programmer from resource limitations in the underlying hardware. For example, most imperative languages support an unlimited number of local variables, "virtualizing" the underlying register file though the use of stack memory and register allocation. Similarly, wide integer operations are often implemented as sequences

of narrow integer operations linked by the propagation of carry bits. As with conventional compilers, we are interested in providing a "seamless" semantic abstraction to the user, but are willing to tolerate some performance degradation when basic hardware limits are exceeded.

Perhaps the most obvious virtualization targets for the Greedy CAM prototype are (1) the number of operations in the body DAG and the degree of interconnection between them, and (2) the number of tables that may participate in a single query. The first case does not appear to be daunting. If a kernel body becomes overly complex; e.g., there are too many operators, or the hardware cannot provide all of the inter-operator connections requested, the body can be partitioned, with intermediate results passed between multiple "implementation kernels" via global storage. While the problem space is known to be exponential, researchers in the graphics domain have constructed useful approximate solutions for the partitioning of fragment shaders (functional DAGs) [CNS+02, FHH04]. It is reasonable to expect that a similar approach could be applied to Greedy CAM kernel bodies.

The second case, virtualizing the number of tables participating in a query, is more problematic. The basic strategy is to partition the kernel into several kernels, each responsible for executing a subset of the initial kernel's `query` specification, such that all dependences between the initial kernel's `bind` operations are honored. This is similar to how databases [Cha98] handle complex queries, decomposing them into trees or DAGs of simpler queries. Making this work requires that we minimize the size of intermediate tables, and that we maximally avoid intermediate memory access and computation that will merely be pruned out by consuming kernels. Unfortunately, we lack two of the tools available to database query optimizers: (1) dynamic information on the size of query results and the selectivity of query operations, and (2) the ability to pipeline intermediate data between queries. Performing query optimization in a purely static manner may require assistance from the programmer.

Another, major, problem with partitioning of queries is deletion. Because deletion is delayed until a complete query tuple has been constructed, the kernel that completes the tuple must somehow communicate this fact back to the kernel(s) producing its input—but this will be too late, as the input kernels will already have terminated. It may appear that this can be solved, at some cost, by having the final kernels construct a list of tuples to be deleted, and traversing this list explicitly (i.e., breaking the recursive dependence by delay). However, doing so can violate the prototype deletion semantics, which requires eager deletion as soon as a query result tuple is formed. We do not see any obvious way around this, other than changing the deletion semantics as described in Section 6.3 so that all deletions are delayed until a kernel completes.

### 6.8.2 Relational queries

The other high-level abstraction we have considered is that of a fully-relational model, in which all columns of all tables involved in a query can participate in the query's constraints. Implementing this requires that we extend the Greedy CAM model to support non-indexed searching (as described in Section 6.2), implement an automatic relational-to-indexed data translation, or both. While database index selection for mixed index/search systems has been explored and implemented [CN97], we are not aware of algorithms supporting purely-indexed systems such as the Greedy CAM prototype. Some of our test programs were first written in a version of LINT without indexing; whole we have evaluated their functional correctness via an interpreter, and have handle-translated many of them to the index-based LINT described in this article, we have not yet explored translation (automatic or user-guided) between the two languages.

# 7 Related Work

The Greedy CAM architecture shares a number of features with existing coarse-grained dataflow and reconfigurable hardware approaches; these correspondences are described in detail in [Bit07]. In this section, we look specifically at programming abstractions based on a streams-and-kernels model and/or explicit query processing.

The Imagine [KDK+02] and Merrimac [DH03] stream processors expose a stream-and-kernel abstraction via the low-level imperative languages `KernelC` and `StreamC`. A `KernelC` program describes processing within

individual kernels, including a form of data-dependent stream I/O, and is compiled to a SIMD controller program. StreamC allocates stream storage and invokes kernels; it is compiled to a conventional ISA for a control processor that drives the kernel and communication hardware.

Brook [Buc03, BFH+04] is a slightly higher level language that provides functional constructs for describing elementwise kernels (i.e., the functions to be mapped or reduced over data streams). Some amount of explicit indexing (i.e., "gather" and "scatter" operations) is provided, but intra-kernel control flow is highly restricted, and cross-invocation local state is not available. Kernel invocation is driven by function calls in a C-like control program. Brook programs can be compiled to `KernelC/StreamC` programs, or (due to their restricted nature) to a combination of C and GPU code.

In comparison with the Greedy CAM prototype, both of these systems execute a conventional imperative program (on a control processor or a CPU host); thus, they may be able to support more complex inter-kernel control and synchronization operations than the Greedy CAM signaling scheme. On the other hand, the Greedy CAM query mechanism supports a more complex range of data access patterns, and natively supports sparse data structures without the use of explicit indirection tables.

The StreamIt language is intended for high-performance dense synchronous streaming applications, for example signal processing. Thus, complex data structures and queries are not emphasized. The basic construct is the *filter*, which reads and writes values into FIFO queues (streams). Filters are linked together using explicit combinators such as `Pipeline`, `SplitJoin` and `FeedbackLoop`. This restrictive programming paradigm enables a large amount of compile-time optimization but does not admit dynamic matching at synchronization.

The Linda [Gel85] and Datalog [CGT89] programming paradigms both make use of query abstractions. Linda is a library that provides multiple "tuple spaces" similar to Greedy CAM tables, but its queries are restricted to returning (and optionally deleting) single tuples. While this approach has proven quite useful for communication and synchronization in distributed systems, the lack of "join" functionality limits its use in implementing complex queries.[29] Datalog is a query language that allows complex queries through the use of logical variables. Like Linda and SQL, it is not a complete programming language, and thus relies on a host language to provide control flow, arithmetic operations, synchronization, etc. The Greedy CAM model prototype explicitly avoids the use of such a host language to ensure that entire programs can be compiled to specialized hardware.

# 8 Conclusion

We have presented our experience with a prototype instance of the Greedy CAM programming model. We have constructed an intermediate language, LINT, that faithfully captures the semantics of the prototype, and have used LINT and its interpreter to explore a number of algorithms and programming idioms within the prototype programming model. This experimentation has led to several insights concerning the programming model, which we expect will influence future Greedy CAM designs. We look forward to the forthcoming low-level hardware simulator, which will allow us to more accurately judge tradeoffs in algorithm design and implementation.

# References

[BFH+04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *ACM SIGGRAPH*, pages 777–786, 2004.

[Bit07] Ray Bittner. Overcoming memory latency and enabling parallelism with the Greedy CAM architecture. Technical Report MSR-TR-2007-10, Microsoft Research, 2007.

[29] One could, of course, write multilevel queries using loop nests in the host language, but this negates the possibility of moving complex, multi-tuple query operations into specialized low-level hardware.

[Ble90] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[Buc03] Ian Buck. Data parallel computation on graphics hardware. In *Graphics Hardware*, 2003.

[CBZ90] Siddhartha Chatterjee, Guy E. Blelloch, and Marco Zagha. Scan primitives for vector computers. In *Supercomputing*, pages 666–675, 1990.

[CGT89] C. Ceri, G. Gottlieb, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.

[Cha98] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Principles of database systems*. ACM SIGACT-SIGMOD-SIGART, 1998.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[CN97] Surajit Chaudhuri and Vivek Narasayya. An efficient, cost-driven index selection tool for microsoft SQL server. In *Proceedings of the 23rd VLDB Conference*, 1997.

[CNS+02] E. Chan, R. Ng, P. Sen, K. Proudfoot, and P. Hanrahan. Efficient partitioning of fragment shaders for multipass rendering on programmable graphics hardware. In *ACM SIGGRAPH*, pages 69–78, 2002.

[DH03] W. Dally and P. Hanrahan. Merrimac: Supercomputing with streams. In *Supercomputing '03*, November 2003.

[FHH04] Tim Foley, Mike Houston, and Pat Hanrahan. Efficient partitioning of fragment shaders for multiple-output hardware. In *ACM SIGRGRAPH/EUROGRAPHICS conference on Graphics Hardware*, pages 45–53, 2004.

[Gel85] David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1), 1985.

[HS86] W. Daniel Hillis and Guy L. Jr. Steele. Data parallel algorithms. *Communications of the ACM*, 1986.

[Kaj07] James Kajiya. Accelerator numeric format specification. In preparation, 2007.

[KDK+02] Ujval J. Kapasi, William J. Dally, Brucek Khailany, John D. Owens, and Scott Rixner. The imagine stream processor. In *IEEE International Conference on Computer Design*, pages 282–288, September 2002.

[Lam88] Monica Lam. Software pipelining: an effective scheduling technique for VLIW machines. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328, 1988.

```
(program vec-add-iterative
  (typedefs
    (typedef _tag (int 8))
    (typedef _void (int 0)))
  (tables
    (table n function _tag _void _void ((0 0 0)))
    (table a function _tag _void real ...) ;; indices 0,1,...,63
    (table b function _tag _void real ...) ;; indices 0,1,...,63
    (table c function _tag _void real ()))
  (kernels
    (kernel add
      (wait)
      (on-entry)
      (read
        (bind n n exists delete ())
        (bind a a exists preserve ((= (path _tag) (n i (path _tag)))))
        (bind b b exists preserve ((= (path _tag) (n i (path _tag))))))
      (body
        (begin
          (cwrite 1 c (table-index n) 0 (+ (table-data a) (table-data b)))
          (cwrite 1 n (+ (table-index n) 1) 0 0)))
      (on-success)
      (on-failure)))
  (schedule (while add)))
```

Figure 10: Iterative vector addition, with explicit induction variable

# Appendix

## 8.1 Sample Programs

### 8.1.1 Vector addition

Dense vector addition is a straightforward elementwise mapping operation. We present four versions, only one of which fits the architecture well.

- `vec-add-iterative` (Fig. 10): The most direct port from a sequential language. The kernel loops over an explicit index variable, which is used to dereference the input arrays and and to indicate the position of the write into the result array.
- `vec-add-iterative2` (Fig. 11): Also an iterative implementation, but avoids the explicit index variable. Note the use of deletion to ensure that queries in distinct iterations (kernel instantiations) will return unique results.
- `vec-add-iterative-by4` (Fig. 12): Same as above, but reads size-4 tiles, reducing the number of kernel initializations required. Note that only vector lengths divisible by 4 are handled correctly—a more realistic version would require a "by-1" cleanup kernel.
- `vec-add-streaming` (Fig. 13): This is the native mapping idiom for the Greedy CAM architecture. Only one kernel initialization is required, and memory access and computation are pipelined.

Figure 14 shows the information collected by the interpreter when the examples are executed on a pair of 64-element input vectors. All versions execute an empty "start-the-world" kernel introduced by the parser. In the first two versions, the `add` kernel initialized 65 times. With the exception of the final execution, which fails immediately, each kernel succeeds in reading a single pair of input elements, writing one output element, and then fails. The third version reduces the loop's trip count (thus the kernel's initialization count) by processing 4 elements per kernel activation. The fourth version accomplishes the entire operation in a single kernel activation. Note that the iterative versions require an additional kernel activation to discover

```
(program vec-add-iterative2
  (typedefs
    (typedef _tag (int 8))
    (typedef _void (int 0)))
  (tables
    (table a function _tag _void real ...) ;; indices 0,1,...,63
    (table b function _tag _void real ...) ;; indices 0,1,...,63
    (table c function _tag _void real ()))
  (kernels
    (kernel add
      (wait)
      (on-entry)
      (read
        (bind a a exists delete ())
        (bind b b exists preserve ((= (path _tag) (a i (path _tag))))))
      (body
        (cwrite 1 c (table-index a) 0 (+ (table-data a) (table-data b))))
      (on-success)
      (on-failure)))
  (schedule (while add)))
```

Figure 11: Iterative vector addition, without induction variable

```
(program vec-add-iterative-by4
  (typedefs
    (typedef _tag (int 8))
    (typedef _void (int 0)))
  (tables
    (table a function _tag _void real ...)
    (table b function _tag _void real ...)
    (table c function _tag _void real ()))
  (kernels
    (kernel add
      (wait)
      (on-entry)
      (read
        (bind a0 a exists delete   ((= 1:0 0)))
        (bind a1 a exists preserve ((= 7:2 (a0 i 7:2)) (= 1:0 1)))
        (bind a2 a exists preserve ((= 7:2 (a0 i 7:2)) (= 1:0 2)))
        (bind a3 a exists preserve ((= 7:2 (a0 i 7:2)) (= 1:0 3)))
        (bind b0 b exists preserve ((= 7:2 (a0 i 7:2)) (= 1:0 0)))
        (bind b1 b exists preserve ((= 7:2 (a0 i 7:2)) (= 1:0 1)))
        (bind b2 b exists preserve ((= 7:2 (a0 i 7:2)) (= 1:0 2)))
        (bind b3 b exists preserve ((= 7:2 (a0 i 7:2)) (= 1:0 3))))
      (body
        (begin
          (cwrite 1 c (table-index a0) 0 (+ (table-data a0) (table-data b0)))
          (cwrite 1 c (table-index a1) 0 (+ (table-data a1) (table-data b1)))
          (cwrite 1 c (table-index a2) 0 (+ (table-data a2) (table-data b2)))
          (cwrite 1 c (table-index a3) 0 (+ (table-data a3) (table-data b3)))))
      (on-success)
      (on-failure)))
  (schedule (while add)))
```

Figure 12: Iterative vector addition, x4 tiled

```
(program vec-add-streaming
  (typedefs
    (typedef _tag (int 8))
    (typedef _void (int 0)))
  (tables
    (table a function _tag _void real ...)
    (table b function _tag _void real ...)
    (table c function _tag _void real ()))
  (kernels
    (kernel add
      (wait)
      (on-entry)
      (read
        (bind a a forall preserve ())
        (bind b b exists preserve ((= (path _tag) (a i (path _tag))))))
      (body
        (cwrite 1 c (table-index a) 0 (+ (table-data a) (table-data b))))
      (on-success)
      (on-failure)))
  (schedule add)))
```

Figure 13: Streaming vector addition

| vec-add-iterative | init | step | bkt-init | read | write | delete | succeed | fail | signal |
|---|---|---|---|---|---|---|---|---|---|
| program-root-128 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| add | 65 | 129 | 193 | 193 | 128 | 64 | 64 | 1 | 64 |
| total | 65 | 130 | 193 | 193 | 128 | 64 | 64 | 66 | 129 |
| vec-add-iterative2 | init | step | bkt-init | read | write | delete | succeed | fail | signal |
| program-root-135 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| add | 65 | 129 | 128 | 128 | 64 | 64 | 64 | 1 | 64 |
| total | 65 | 130 | 128 | 128 | 64 | 64 | 64 | 2 | 65 |
| vec-add-iterative-by4 | init | step | bkt-init | read | write | delete | succeed | fail | signal |
| program-root-141 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| add | 17 | 33 | 128 | 128 | 64 | 16 | 16 | 1 | 16 |
| total | 17 | 34 | 128 | 128 | 64 | 16 | 16 | 2 | 17 |
| vec-add-streaming | init | step | bkt-init | read | write | delete | succeed | fail | signal |
| program-root-143 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| add | 1 | 65 | 128 | 128 | 64 | 0 | 1 | 0 | 0 |
| total | 1 | 66 | 128 | 128 | 64 | 0 | 1 | 1 | 1 |

Figure 14: Vector addition statistics (64-element vector)

```
(program vec-red-iterative
  (typedefs
    (typedef _tag (int 8))
    (typedef _void (int 0)))
  (tables
    (table x function _tag _void real ...)
    (table y function _tag _void real ((0 0 0.0))))
  (kernels
    (kernel red-loop
      (wait)
      (on-entry)
      (read
        (bind x x exists delete ())
        (bind y y exists delete ()))
      (body
        (cwrite 1 y 0 0 (+ (table-data y) (table-data x))))
      (on-success)
      (on-failure)))
  (schedule (while red-loop)))
```

Figure 15: Iterative vector reduction

that the input has been exhausted; e.g., the failure terminates the loop. The streaming kernel does not require an additional activation.

### 8.1.2 Vector reduction

This section shows three implementations of vector reduction:

- `vec-red-iterative` (Fig. 15): This is the standard iterative algorithm that adds each input value into an accumulator. Because the Greedy CAM model does not support write-read dependences within a single kernel activation, we are forced to restart the kernel after each accumulator update.
- `vec-red-iterative-by4` (Fig. 16): This is the iterative algorithm, unrolled by a factor of four. This improves both the kernel initialization count and the number of reads and writes to the accumulator. Note that the input length must be a multiple of the tile size (4, in this case). Compensation code for non-by4 inputs would have the same form as the loop in `vec-red-iterative`.
- `vec-red-tree` (Fig. 17): This version is able to stream across the entire input, but is unable to perform the complete accumulation in a single pass.

Figure 18 gives the statistics for reduction of a 64-element vector. While tiling is clearly a good idea, the choice between tiled iteration and reduction trees is less obvious. While the tree version executes slightly more memory operations, a higher fraction of them run in streaming mode, and there are fewer kernel initializations. As input size increases, the tree version will become more advantageous because its kernel instantiation count grows logarithmically with input size, whereas the iterative version's will grow linearly.

### 8.1.3 Matrix multiplication

This section describes a number of matrix multiplication algorithms:

- `mat-mult-separate` (Fig. 19): This version performs all of the multiplications in a single kernel activation, generating a three-dimensional intermediate result that is then reduced along the $j$ dimension via a 4-wide tree reduction algorithm.
- `mat-mult-combined` (Fig. 20): This version combines multiplication and accumulation by processing one $j$ index at a time, streaming over the $i$ and $k$ indices.

```
(program vec-red-iterative-by4
  (typedefs
    (typedef _tag (int 8))
    (typedef _void (int 0)))
  (tables
    (table x function _tag _void real ...)
    (table y function _tag _void real ((0 0 0.0))))
  (kernels
    (kernel red-loop-by4
      (wait)
      (on-entry)
      (read
        (bind x0 x exists delete ((= 1:0 0)))
        (bind x1 x exists delete ((= 7:2 (x0 i 7:2)) (= 1:0 1)))
        (bind x2 x exists delete ((= 7:2 (x0 i 7:2)) (= 1:0 2)))
        (bind x3 x exists delete ((= 7:2 (x0 i 7:2)) (= 1:0 3)))
        (bind y y exists delete ()))
      (body
        (let* ((sum0 (+ (table-data x0) (table-data x1)))
               (sum1 (+ (table-data x2) (table-data x3)))
               (sum2 (+ sum0 sum1)))
          (cwrite 1 y 0 0 (+ (table-data y) sum2))))
      (on-success)
      (on-failure)))
  (schedule (while red-loop-by4)))
```

Figure 16: Iterative vector reduction, x4 tiled

```
(program vec-red-tree
  (typedefs
    (typedef _tag (int 8))
    (typedef _void (int 0)))
  (tables
    (table x function _tag _void real ...)
  (kernels
    (kernel red-by4
      (wait)
      (on-entry)
      (read
        (bind x0 x forall delete ((= 1:0 0)))
        (bind x1 x exists delete ((= 7:2 (x0 i 7:2)) (= 1:0 1)))
        (bind x2 x exists delete ((= 7:2 (x0 i 7:2)) (= 1:0 2)))
        (bind x3 x exists delete ((= 7:2 (x0 i 7:2)) (= 1:0 3))))
      (body
        (let* ((sum0 (+ (table-data x0) (table-data x1)))
               (sum1 (+ (table-data x2) (table-data x3)))
               (sum2 (+ sum0 sum1)))
          (cwrite 1 x (shr (table-index x0) 2) 0 sum2)))
      (on-success)
      (on-failure)))
  (schedule (while red-by4)))
```

Figure 17: Tree vector reduction, x4 tiled

| vec-red-iterative | init | step | bkt-init | read | write | delete | succeed | fail | signal |
|---|---|---|---|---|---|---|---|---|---|
| program-root-60 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| red-loop | 65 | 129 | 128 | 128 | 64 | 128 | 64 | 1 | 64 |
| total | 65 | 130 | 128 | 128 | 64 | 128 | 64 | 2 | 65 |
| vec-red-iterative-by4 | init | step | bkt-init | read | write | delete | succeed | fail | signal |
| program-root-66 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| red-loop-by4 | 17 | 33 | 80 | 80 | 16 | 80 | 16 | 1 | 16 |
| total | 17 | 34 | 80 | 80 | 16 | 80 | 16 | 2 | 17 |
| vec-red-tree | init | step | bkt-init | read | write | delete | succeed | fail | signal |
| program-root-84 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| red-by4 | 4 | 25 | 85 | 85 | 21 | 84 | 3 | 1 | 3 |
| total | 4 | 26 | 85 | 85 | 21 | 84 | 3 | 2 | 4 |

Figure 18: Vector reduction statistics (64-element vector)

- `mat-mult-separate-tiled1` (Figs. 21 and 22): Performs the expansion pass with 2x2 tiling, which enables maximal computation per query (four multiplications and four additions). This halves the size of the $j$ dimension in the subsequent x4 tree reduction.
- `mat-mult-separate-tiled2` (Figs. 23 and 24): Performs the expansion pass with 1x4 tiling on the $a$ input, and 4x1 tiling on the $b$ input. This accomplishes one less addition per query then `mat-mult-separate-tiled1` , but reduces the size of the $j$ dimension by a factor of 4 (rather than 2) in the subsequent x4 tree reduction.
- `mat-mult-combined-tiled` (Fig. 25): This is the combined algorithm, using the 1x4, 4x1 input tiling, which reduces the iteration count by a factor of 4.[30]

Statistics for 16x16 case are shown in Figure 26. We see that tiling greatly reduces memory writes, while combining the expansion and reduction phases saves intermediate storage. Combining the two yields the best results w.r.t. memory operation counts, though the additional benefit from larger stream lengths in the `mat-mult-separate-tiled2` may overcome the cost of additional memory operations.

#### 8.1.4 Sparse vector addition

We show two approaches to sparse mapping, which differ in their treatment of values that appear only in one input.

- `sparse-vec-add-par` (Fig. 27): The "present in a and b," "present in a only," and "present in b only" cases run in parallel. The "only" kernels use negation to establish that no corresponding element is available in the other input vector.
- `sparse-vec-add-2phase` (Fig. 28): The "a and b" case runs first, deleting all matching inputs. The remaining cases can now run in parallel.

The first version performs more memory read operations because, for each input position, establishing negation requires attempting to read the corresponding entry of the other input. However, if sufficient memory bandwidth is available, it will complete more quickly than the second version, which must wait to process singleton inputs until all of the paired inputs have been processed.

This algorithm is the only example that makes use of parallel scheduling of kernels. While most of the examples are indeed parallelizable (see Section 5), we have chosen not to present parallel implementations

[30]The 2x2 input tiling was not used with the "combined" algorithm because doing so requires 12 query bindings, exceeding the capacity of the initial hardware implementation.

```
(program mat-mult-separate
  (typedefs
    (typedef _i    (int 8))
    (typedef _j    (int 8))
    (typedef _k    (int 8))
    (typedef _ik   (struct (i (int 8)) (k (int 8))))
    (typedef _ikj  (struct (i (int 8)) (k (int 8)) (j (int 8))))
    (typedef _void (int 0))
    (typedef _elt  real))
  (tables
    (table a relation _i _j _elt ...)
    (table b relation _j _k _elt ...)
    (table e relation _ikj _void _elt
      ())
    (table c function _ik _void _elt
      ()))
  (kernels
    (kernel expand
      (wait)
      (on-entry)
      (read
        (bind a a forall preserve ())                  ;; read all rows of a
        (bind b b forall preserve ((= (path _j) (a e (path _j))))) ;; index b with a's ephemeral data only
      (body
        (let ((product (* (table-data a) (table-data b)))
              (i (table-index a))
              (j (table-ephemeral a))
              (k (table-ephemeral b)))
          (let* ((ikjtag i)
                 (ikjtag (inject ikjtag (path _ikj k) k))
                 (ikjtag (inject ikjtag (path _ikj j) j)))
            (cwrite 1 e ikjtag 0 product))))            ;; populate expansion
      (on-success)
      (on-failure))
    ;; red-loop is x4 streamized
    (kernel red-loop
      (wait)
      (on-entry)
      (read
        (bind e0 e forall delete ((= 17:16 0)))
        (bind e1 e exists delete ((= 17:16 1) (= 23:18 (e0 i 23:18)) (= 15:0 (e0 i 15:0))))
        (bind e2 e forall delete ((= 17:16 2) (= 23:18 (e1 i 23:18)) (= 15:0 (e1 i 15:0))))
        (bind e3 e exists delete ((= 17:16 3) (= 23:18 (e2 i 23:18)) (= 15:0 (e2 i 15:0))))
        )
      (body
        (let ((r (+ (+ (table-data e0) (table-data e1))
                    (+ (table-data e2) (table-data e3))))
              (ik (project 15:0 (table-index e0)))
              (j (project 23:16 (table-index e0))))
          (let ((index (inject ik 21:16 (project 15:2 j))))
            (cwrite 1 e index 0 r))))
      (on-success)
      (on-failure)))
  (schedule (begin
              expand
              (while
                red-loop))))
```

Figure 19: Matrix multiplication with expansion and reduction phases

```
(program mat-mult-combined
  (typedefs
    (typedef _i    (int 8))
    (typedef _j    (int 8))
    (typedef _ij   (struct (i (int 8)) (j (int 8))))
    (typedef _jk   (struct (j (int 8)) (k (int 8))))
    (typedef _ik   (struct (i (int 8)) (k (int 8))))
    (typedef _void (int 0))
    (typedef _elt  real))
  (tables
    (table a relation _ij _void _elt ...)
    (table b relation _jk _void _elt ...)
    (table c function _ik _void _elt ...)
    (table js function _j _void _void ((0 0 0))))
  (kernels
    (kernel mul-add
      (wait)
      (on-entry)
      (read
        (bind js js exists preserve ())
        (bind a a forall preserve ((= (path _ij j) (js i (path _j)))))
        (bind b b forall preserve ((= (path _jk j) (js i (path _j)))))
        (bind c c exists delete ((= (path _ik i) (a i (path _ij i)))
                                 (= (path _ik k) (b i (path _jk k))))))
      (body
        (let ((sum (+ (tref c d) (* (tref a d) (tref b d)))))
          (cwrite 1 c (tref c i) 0 sum)))
      (on-success)
      (on-failure))
    (kernel inc-js
      (wait)
      (on-entry)
      (read
        (bind js js exists delete ()))
      (body
        (let ((j (tref js i)))
          (cwrite 1 js (+ j 1) 0 0)))
      (on-success)
      (on-failure)))
  (schedule (while mul-add inc-js)))
```

Figure 20: Matrix multiplication, combining expansion and reduction

```
(program mat-mul-separate-tiled1
  (typedefs
    (typedef _index2 (struct (outer (int 8)) (inner (int 8))))
    (typedef _index3 (struct (i (int 8)) (k (int 8)) (j (int 8))))
    (typedef _dim  (int 8))
    (typedef _ik   (struct (i (int 8)) (k (int 8))))
    (typedef _void (int 0))
    (typedef _elt  real))
  (tables
    (table a function _index2 _void _elt ...)
    (table b function _index2 _void _elt ...)
    (table e function _index3 _void _elt ()))
  (kernels
    (kernel expand
      (wait)
      (on-entry)
      (read
        (bind a00 a forall preserve ((= 0:0 0) (= 8:8 0)))
        (bind a01 a exists preserve ((= 0:0 0) (= 8:8 1) (= 15:9 (a00 i 15:9)) (= 7:1 (a00 i 7:1))))
        (bind a10 a exists preserve ((= 0:0 1) (= 8:8 0) (= 15:9 (a01 i 15:9)) (= 7:1 (a01 i 7:1))))
        (bind a11 a exists preserve ((= 0:0 1) (= 8:8 1) (= 15:9 (a10 i 15:9)) (= 7:1 (a10 i 7:1))))
        (bind b00 b forall preserve ((= 7:1 (a11 i 15:9)) (= 0:0 0) (= 8:8 0)))
        (bind b01 b exists preserve ((= 0:0 0) (= 8:8 1) (= 15:9 (b00 i 15:9)) (= 7:1 (b00 i 7:1))))
        (bind b10 b exists preserve ((= 0:0 1) (= 8:8 0) (= 15:9 (b01 i 15:9)) (= 7:1 (b01 i 7:1))))
        (bind b11 b exists preserve ((= 0:0 1) (= 8:8 1) (= 15:9 (b10 i 15:9)) (= 7:1 (b10 i 7:1)))))
      (body
        (let ((c00 (+ (* (table-data a00) (table-data b00)) (* (table-data a01) (table-data b10))))
              (c01 (+ (* (table-data a00) (table-data b01)) (* (table-data a01) (table-data b11))))
              (c10 (+ (* (table-data a10) (table-data b00)) (* (table-data a11) (table-data b10))))
              (c11 (+ (* (table-data a10) (table-data b01)) (* (table-data a11) (table-data b11))))
              (i (project (path _index2 outer) (table-index a00)))
              (j (project (path _index2 inner) (table-index a00)))
              (k (project (path _index2 inner) (table-index b00))))
          (let* ((ikj (inject 0 (path _index3 i) i))
                 (ikj (inject ikj (path _index3 k) k))
                 (ikj (inject ikj (path _index3 j) (project 7:1 j))))
            (cwrite 1 e (inject (inject ikj 0:0 0) 8:8 0) 0 c00)
            (cwrite 1 e (inject (inject ikj 0:0 0) 8:8 1) 0 c01)
            (cwrite 1 e (inject (inject ikj 0:0 1) 8:8 0) 0 c10)
            (cwrite 1 e (inject (inject ikj 0:0 1) 8:8 1) 0 c11))))
      (on-success)
      (on-failure))
```

Figure 21: `mat-mult-separate1`, expansion kernel, tiled x2 on all dimensions

```
  ;; red-loop is x4 streamized
  (kernel red-loopx4
    (wait)
    (on-entry)
    (read
      (bind e0 e forall delete ((= 17:16 0)))
      (bind e1 e exists delete ((= 17:16 1) (= 23:18 (e0 i 23:18)) (= 15:0 (e0 i 15:0))))
      (bind e2 e exists delete ((= 17:16 2) (= 23:18 (e0 i 23:18)) (= 15:0 (e0 i 15:0))))
      (bind e3 e exists delete ((= 17:16 3) (= 23:18 (e0 i 23:18)) (= 15:0 (e0 i 15:0)))))
    (body
      (let ((r (+ (+ (table-data e0) (table-data e1))
                  (+ (table-data e2) (table-data e3))))
            (ik (project 15:0 (table-index e0)))
            (new-j (project 23:18 (table-index e0))))
        (let ((index (inject ik 21:16 new-j)))
          (cwrite 1 e index 0 r))))
    (on-success)
    (on-failure))
  (kernel red-loopx2
    (wait)
    (on-entry)
    (read
      (bind e0 e forall delete ((= 17:16 0)))
      (bind e1 e exists delete ((= 17:16 1) (= 23:18 (e0 i 23:18)) (= 15:0 (e0 i 15:0)))))
    (body
      (let ((r (+ (table-data e0) (table-data e1)))
            (ik (project 15:0 (table-index e0)))
            (new-j (project 23:17 (table-index e0))))
        (let ((index (inject ik 21:17 new-j)))
          (cwrite 1 e index 0 r))))
    (on-success)
    (on-failure)))
(schedule (begin
            expand
            (while red-loopx4)
            red-loopx2)))
```

Figure 22: `mat-mult-separate1`, reduction kernels

```
(define mat-mult-separate-tiled2
  `(program mat-mult-separate-tiled2
     (typedefs
       (typedef _index2 (struct (outer (int 8)) (inner (int 8))))
       (typedef _index3 (struct (i (int 8)) (k (int 8)) (j (int 8))))
       (typedef _dim  (int 8))
       (typedef _ij   (struct (i (int 8)) (j (int 8))))
       (typedef _jk   (struct (j (int 8)) (k (int 8))))
       (typedef _ijk   (struct (i (int 8)) (k (int 8)) (j (int 8))))
       (typedef _void (int 0))
       (typedef _elt  real))
     (tables
       (table a function _index2 _void _elt ...)
       (table b function _index2 _void _elt ...)
       (table e function _index3 _void _elt ()))
     (kernels
       (kernel expand
         (wait)
         (on-entry)
         (read
           (bind a0 a forall preserve ((= 9:8 0)))
           (bind a1 a exists preserve ((= 15:10 (a0 i 15:10))
                                       (= 9:8 1)
                                       (= (path _ij i) (a0 i (path _ij i)))))
           (bind a2 a exists preserve ((= 15:10 (a0 i 15:10))
                                       (= 9:8 2)
                                       (= (path _ij i) (a0 i (path _ij i)))))
           (bind a3 a exists preserve ((= 15:10 (a0 i 15:10))
                                       (= 9:8 3)
                                       (= (path _ij i) (a0 i (path _ij i)))))

           (bind b0 b forall preserve ((= 7:2 (a0 i 15:10))
                                       (= 1:0 0)))
           (bind b1 b exists preserve ((= (path _jk k) (b0 i (path _jk k)))
                                       (= 7:2 (b0 i 7:2))
                                       (= 1:0 1)))
           (bind b2 b exists preserve ((= (path _jk k) (b0 i (path _jk k)))
                                       (= 7:2 (b0 i 7:2))
                                       (= 1:0 2)))
           (bind b3 b exists preserve ((= (path _jk k) (b0 i (path _jk k)))
                                       (= 7:2 (b0 i 7:2))
                                       (= 1:0 3))))
         (body
           (let* ((prod0 (* (tref a0 d) (tref b0 d)))
                  (prod1 (* (tref a1 d) (tref b1 d)))
                  (prod2 (* (tref a2 d) (tref b2 d)))
                  (prod3 (* (tref a3 d) (tref b3 d)))
                  (sum (+ (+ prod0 prod1) (+ prod2 prod3)))
                  (i (project (path _ij i) (table-index a0)))
                  (j (project (path _ij j) (table-index a0)))
                  (k (project (path _jk k) (table-index b0)))
                  (nj (shr j 2))
                  (nij (inject i (path _ijk j) nj))
                  (nijk (inject nij (path _ijk k) k)))
             (cwrite 1 e nijk 0 sum)))
         (on-success)
         (on-failure))
```

Figure 23: `mat-mult-separate`, expansion kernel tiled x4 on $j$ dimension

```
;; red-loop is x4 streamized
(kernel red-loopx4
  (wait)
  (on-entry)
  (read
    (bind e0 e forall delete ((= 17:16 0)))
    (bind e1 e exists delete ((= 17:16 1) (= 23:18 (e0 i 23:18)) (= 15:0 (e0 i 15:0))))
    (bind e2 e exists delete ((= 17:16 2) (= 23:18 (e0 i 23:18)) (= 15:0 (e0 i 15:0))))
    (bind e3 e exists delete ((= 17:16 3) (= 23:18 (e0 i 23:18)) (= 15:0 (e0 i 15:0)))))
  (body
    (let ((r (+ (+ (table-data e0) (table-data e1))
                (+ (table-data e2) (table-data e3))))
          (ik (project 15:0 (table-index e0)))
          (new-j (project 23:18 (table-index e0))))
      (let ((index (inject ik 21:16 new-j)))
        (cwrite 1 e index 0 r))))
  (on-success)
  (on-failure)))
(schedule (begin
            expand
            (while red-loopx4)))))
```

Figure 24: `mat-mult-separate2`, reduction kernel

here because of the large volume of code involved; i.e., a large number of kernels are replicated with only a few changes to the queries (e.g., small constant constraints against low-order bit values), plus summarization code to combine the results of multiple parallel reduction phases.

### 8.1.5 Sparse matrix multiplication

As was the case with dense matrix multiplication, we can choose to separate the expansion and reduction operations, or we can combine them:

- `sparse-mat-mult-separate` (Fig. 30): The expansion kernel streams across all three dimensions, and thus need be activated only once. In addition to performing multiplications, this kernel also computes a list of all $j$ indices for which at least one product has been computed (note the use of a "function" table to ensure that each $j$ value is recorded at most once). This list then drives the per-plane iterative reduction kernel; tree-based reduction is not an option here because the expansion result is sparse.

- `sparse-mat-mult-combined` (Fig. 31): This version mimics the dense algorithm `mat-mult-combined` in computing one plane of multiplication at a time, updating the accumulator plane appropriately. Unfortunately, we don't know in advance which locations in the accumulator will eventually be updated. Zeroing the entire accumulator would result in a dense result, which is undesirable if, for example, we want to perform a series of multiplications. Thus, this code first performs the full expansion phase (minus the actual multiplications) to zero the appropriate accumulator elements.[31]

As we can see from the code, or from the statistics (see Figure 32), both approaches perform the identical query in their initial kernels, but the "separate" version also accomplishes multiplication at this time. This halves the number of input matrix reads in the accumulation kernel, so we expect that the "separate" algorithm will always outperform its "combined" counterpart.

[31] Alternately, we could statically zero the entire accumulator, run the combined kernel, then remove all zeros from the output. However, since we expect that a sparse matrix will have a majority of zero values, this is likely to be more expensive than the selective initialization approach.

```
(program mat-mult-combined-tiled
  (typedefs
    (typedef _j    (int 8))
    (typedef _ij   (struct (i (int 8)) (j (int 8))))
    (typedef _jk   (struct (j (int 8)) (k (int 8))))
    (typedef _ik   (struct (i (int 8)) (k (int 8))))
    (typedef _void (int 0))
    (typedef _elt  real))
  (tables
    (table a relation _ij _void _elt ...)
    (table b relation _jk _void _elt ...)
    (table c function _ik _void _elt ...)
    (table js function _j _void _void ((0 0 0))))
  (kernels
    (kernel mul-add
      (wait)
      (on-entry)
      (read
        (bind js js exists preserve ())
        (bind a0 a forall preserve ((= 15:10 (js i 7:2))
                                    (= 9:8 0)))
        (bind a1 a forall preserve ((= 15:10 (js i 7:2))
                                    (= 9:8 1)
                                    (= (path _ij i) (a0 i (path _ij i)))))
        (bind a2 a forall preserve ((= 15:10 (js i 7:2))
                                    (= 9:8 2)
                                    (= (path _ij i) (a0 i (path _ij i)))))
        (bind a3 a forall preserve ((= 15:10 (js i 7:2))
                                    (= 9:8 3)
                                    (= (path _ij i) (a0 i (path _ij i)))))
        (bind b0 b forall preserve ((= 7:2 (js i 7:2))
                                    (= 1:0 0)))
        (bind b1 b forall preserve ((= 7:2 (js i 7:2))
                                    (= 1:0 1)
                                    (= (path _jk k) (b0 i (path _jk k)))))
        (bind b2 b forall preserve ((= 7:2 (js i 7:2))
                                    (= 1:0 2)
                                    (= (path _jk k) (b0 i (path _jk k)))))
        (bind b3 b forall preserve ((= 7:2 (js i 7:2))
                                    (= 1:0 3)
                                    (= (path _jk k) (b0 i (path _jk k)))))
        (bind c c exists preserve ((= (path _ik i) (a0 i (path _ik i)))
                                   (= (path _jk k) (b0 i (path _jk k))))))

      (body
        (let* ((prod0 (* (tref a0 d) (tref b0 d)))
               (prod1 (* (tref a1 d) (tref b1 d)))
               (prod2 (* (tref a2 d) (tref b2 d)))
               (prod3 (* (tref a3 d) (tref b3 d)))
               (cval (tref c d))
               (sum (+ cval (+ (+ prod0 prod1) (+ prod2 prod3)))))
          (cwrite 1 c (tref c i) 0 sum)))
      (on-success)
      (on-failure))
    (kernel inc-js
      (wait)
      (on-entry)
      (read
        (bind js js exists delete ()))
      (body
        (let ((j (tref js i)))
          (cwrite 1 js (+ j 4) 0 0)))
      (on-success)
      (on-failure)))
  (schedule (while mul-add inc-js)))
```

Figure 25: `mat-mult-combined`, tiled x4 on $j$ dimension

| mat-mult-separate | init | step | bkt-init | read | write | delete | succeed | fail | signal |
|---|---|---|---|---|---|---|---|---|---|
| program-root-143 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| expand | 1 | 4097 | 272 | 4352 | 4096 | 0 | 1 | 0 | 1 |
| red-loop | 3 | 1283 | 5376 | 5376 | 1280 | 5120 | 2 | 1 | 2 |
| total | 4 | 5381 | 5648 | 9728 | 5376 | 5120 | 3 | 2 | 4 |
| mat-mult-combined | init | step | bkt-init | read | write | delete | succeed | fail | signal |
| program-root-148 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| mul-add | 17 | 4113 | 8465 | 8465 | 4096 | 4096 | 16 | 1 | 16 |
| inc-js | 16 | 32 | 16 | 16 | 16 | 16 | 16 | 0 | 16 |
| total | 33 | 4146 | 8481 | 8481 | 4112 | 4112 | 32 | 2 | 33 |
| mat-mult-separate-tiled1 | init | step | bkt-init | read | write | delete | succeed | fail | signal |
| program-root-165 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| expand | 1 | 513 | 2304 | 2304 | 2048 | 0 | 1 | 0 | 1 |
| red-loopx4 | 2 | 514 | 2560 | 2560 | 512 | 2048 | 1 | 1 | 2 |
| red-loopx2 | 1 | 257 | 512 | 512 | 256 | 512 | 1 | 0 | 0 |
| total | 4 | 1285 | 5376 | 5376 | 2816 | 2560 | 3 | 2 | 4 |
| mat-mult-separate-tiled2 | init | step | bkt-init | read | write | delete | succeed | fail | signal |
| program-root-172 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| expand | 1 | 1025 | 4352 | 4352 | 1024 | 0 | 1 | 0 | 1 |
| red-loopx4 | 2 | 258 | 1280 | 1280 | 256 | 1024 | 1 | 1 | 1 |
| total | 3 | 1284 | 5632 | 5632 | 1280 | 1024 | 2 | 2 | 3 |
| mat-mult-combined-tiled | init | step | bkt-init | read | write | delete | succeed | fail | signal |
| program-root-165 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| mul-add | 5 | 1029 | 5381 | 5381 | 1024 | 0 | 4 | 1 | 4 |
| inc-js | 4 | 8 | 4 | 4 | 4 | 4 | 4 | 0 | 4 |
| total | 9 | 1038 | 5385 | 5385 | 1028 | 4 | 8 | 2 | 9 |

Figure 26: Matrix multiplication statistics (16 x 16 matrices)

```
(program sparse-vec-add-par
  (typedefs
    (typedef _tag (int 8))
    (typedef _void (int 0)))
  (tables
    (table a function _tag _void real ...)  ;; 0, 3, 6, ..., 57
    (table b function _tag _void real ...)  ;; 0, 5, 10, ..., 95
    (table c function _tag _void real ()))
  (kernels
    (kernel add-pairs
      (wait)
      (on-entry)
      (read
        (bind a a forall preserve ())                    ;; read all rows of a
        (bind b b exists preserve ((= 7:0 (a i 7:0)))))   ;; index b with a's index only
      (body
        (let ((result (+ (table-data a) (tref b d))))
          (cwrite 1 c (table-index a) 0 result)))
      (on-success)
      (on-failure))
    (kernel add-a-only
      (wait)
      (on-entry)
      (read
        (bind a a forall  preserve ())                   ;; read all rows of a
        (bind b b !exists preserve ((= 7:0 (a i 7:0))))  ;; don't allow a match on b
      (body
        (let ((result (table-data a)))
          (cwrite 1 c (table-index a) 0 result)))
      (on-success)
      (on-failure))
    (kernel add-b-only
      (wait)
      (on-entry)
      (read
        (bind b b forall  preserve ())                   ;; read all rows of b
        (bind a a !exists preserve ((= 7:0 (b i 7:0))))  ;; don't allow a match on a
      (body
        (let ((result (table-data b)))
          (cwrite 1 c (table-index b) 0 result)))
      (on-success)
      (on-failure)))
  (schedule (par done add-pairs add-a-only add-b-only)))
```

Figure 27: `sparse-vec-add-par`

```
(program sparse-vec-add-2phase
  (typedefs
    (typedef _tag (int 8))
    (typedef _void (int 0)))
  (tables
    (table a function _tag _void real ...)  ;; 0, 3, 6, ..., 57
    (table b function _tag _void real ...)  ;; 0, 5, 10, ..., 95
    (table c function _tag _void real ()))
  (kernels
    (kernel add-pairs
      (wait)
      (on-entry)
      (read
        (bind a a forall delete ())                    ;; read all rows of a
        (bind b b exists delete ((= 7:0 (a i 7:0)))))  ;; index b with a's index only
      (body
        (let ((result (+ (table-data a) (tref b d))))
          (cwrite 1 c (table-index a) 0 result)))
      (on-success)
      (on-failure))
    (kernel add-a-only
      (wait)
      (on-entry)
      (read
        (bind a a forall preserve ()))                 ;; read all rows of a
      (body
        (cwrite 1 c (table-index a) 0 (table-data a)))
      (on-success)
      (on-failure))
    (kernel add-b-only
      (wait)
      (on-entry)
      (read
        (bind b b forall preserve ()))                 ;; read all rows of b
      (body
        (cwrite 1 c (table-index b) 0 (table-data b)))
      (on-success)
      (on-failure)))
  (schedule (begin add-pairs (par done add-a-only add-b-only))))
```

Figure 28: `sparse-vec-add-2phase`

| sparse-vec-add-par | init | step | bkt-init | read | write | delete | succeed | fail | signal |
|---|---|---|---|---|---|---|---|---|---|
| program-root-102 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 3 |
| add-pairs | 1 | 4 | 22 | 22 | 3 | 0 | 1 | 0 | 0 |
| add-a-only | 1 | 17 | 22 | 22 | 16 | 0 | 1 | 0 | 0 |
| add-b-only | 1 | 17 | 22 | 22 | 16 | 0 | 1 | 0 | 0 |
| total | 3 | 18 | 66 | 66 | 35 | 0 | 3 | 1 | 3 |
| sparse-vec-add-2phase | init | step | bkt-init | read | write | delete | succeed | fail | signal |
| program-root-108 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| add-pairs | 1 | 4 | 22 | 22 | 3 | 6 | 1 | 0 | 2 |
| add-a-only | 1 | 17 | 16 | 16 | 16 | 0 | 1 | 0 | 0 |
| add-b-only | 1 | 17 | 16 | 16 | 16 | 0 | 1 | 0 | 0 |
| total | 3 | 22 | 54 | 54 | 35 | 6 | 3 | 1 | 3 |

Figure 29: Sparse vector addition

```
(program sparse-mat-mult-separate
  (typedefs
    (typedef _i    (int 8))
    (typedef _j    (int 8))
    (typedef _k    (int 8))
    (typedef _ik   (struct (i (int 8)) (k (int 8))))
    (typedef _void (int 0))
    (typedef _elt  real))
  (tables
    (table a relation _i _j _elt ...) ;;  upper-triangular half of a 16x16 "iota" matrix
    (table b relation _j _k _elt ...) ;;  16x16 matrix that reverses the columns
    (table e relation _j _ik _elt ())
    (table js function _j _void _void ())
    (table c function _ik _void _elt ()))
  (kernels
    (kernel expand
      (wait)
      (on-entry)
      (read
        (bind a a forall preserve ())                  ;; read all rows of a
        (bind b b forall preserve ((= (path _j) (a e (path _j))))) ;; index b with a's ephemeral data only
      (body
        (let ((product (* (table-data a) (table-data b)))
              (i (table-index a))
              (j (table-ephemeral a))
              (k (table-ephemeral b)))
          (let ((iktag (inject i (path _ik k) k)))     ;; compute output tag (concat k i)
            (cwrite 1 e j iktag product)               ;; write to expansion target
            (cwrite 1 c iktag  0 0.0)                  ;; initialize accumulator
            (cwrite 1 js j 0 0))))                     ;; remember that this "j" plane contains data
      (on-success)
      (on-failure))
    (kernel reduce-add
      (wait)
      (on-entry)
      (read
        (bind js js exists delete ())
        (bind e e forall preserve ((= (path _j) (js i (path _j)))))
        (bind c c forall preserve ((= (path _ik) (e e (path _ik))))) ;; delete ok too
      (body
        (let ((sum (+ (tref e d) (tref c d))))
           (cwrite 1 c (tref c i) 0 sum)))             ;; write sum to accum, at read's index value
      (on-success)
      (on-failure)))
  (schedule (begin
              expand
              (while
                reduce-add))))
```

Figure 30: `sparse-mat-mult-separate`

```
(program sparse-mat-mult-combined
  (typedefs
    (typedef _i    (int 8))
    (typedef _j    (int 8))
    (typedef _k    (int 8))
    (typedef _ij   (struct (i (int 8)) (j (int 8))))
    (typedef _jk   (struct (j (int 8)) (k (int 8))))
    (typedef _ik   (struct (i (int 8)) (k (int 8))))
    (typedef _void (int 0))
    (typedef _elt  real))
  (tables
    (table a function _ij _void _elt ...) ;;  upper-triangular half of a 16x16 "iota" matrix
    (table b function _jk _void _elt ...) ;;  16x16 matrix that reverses the columns
    (table c function _ik _void _elt ())
    (table js function _j _void _void ((0 0 0))))
  (kernels
    (kernel init-accum
      (wait)
      (on-entry)
      (read
        (bind a a forall preserve ())
        (bind b b forall preserve ((= (path _jk j) (a i (path _ij j))))))
      (body
        (let ((ci (project (path _ij i) (table-index a)))
              (cj (project (path _ij j) (table-index a)))
              (ck (project (path _jk k) (table-index b))))
          (cwrite 1 c (inject ci (path _ik k) ck) 0 0.0)
          (cwrite 1 js cj 0 0)))
      (on-success)
      (on-failure))
    (kernel mul-add
      (wait)
      (on-entry)
      (read
        (bind js js exists delete ())
        (bind a a forall preserve ((= (path _ij j) (js i (path _j)))))
        (bind b b forall preserve ((= (path _jk j) (js i (path _j)))))
        (bind c c exists delete ((= (path _ik i) (a i (path _ij i)))
                                 (= (path _ik k) (b i (path _jk k))))))
      (body
        (let ((sum (+ (tref c d) (* (tref a d) (tref b d)))))
          (cwrite 1 c (tref c i) 0 sum)))
      (on-success)
      (on-failure)))
  (schedule (begin init-accum (while mul-add))))
```

Figure 31: `sparse-mat-mult-combined`

| sparse-mat-mult-expanding | init | step | bkt-init | read | write | delete | succeed | fail | signal |
|---|---|---|---|---|---|---|---|---|---|
| program-root-115 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| expand | 1 | 137 | 152 | 272 | 408 | 0 | 1 | 0 | 1 |
| reduce-add | 17 | 153 | 168 | 288 | 136 | 136 | 16 | 1 | 16 |
| total | 18 | 291 | 320 | 560 | 544 | 136 | 17 | 2 | 18 |
| sparse-mat-mult-combined | init | step | bkt-init | read | write | delete | succeed | fail | signal |
| program-root-129 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| init-accum | 1 | 137 | 272 | 272 | 272 | 0 | 1 | 0 | 1 |
| mul-add | 17 | 153 | 424 | 424 | 136 | 272 | 16 | 1 | 16 |
| total | 18 | 291 | 696 | 696 | 408 | 272 | 17 | 2 | 18 |

Figure 32: Sparse matrix multiplication statistics

```
(define mst
  '(program mst
     (typedefs
       (typedef _id (int 8))
       (typedef _wt (int 16))
       (typedef _void (int 0))
       (typedef _edge (struct (dest _id) (wt _wt))))    ;; note wt in hi bits
     (tables
       (table r function _id _void _void ((0 0 0)))     ;; contains a single index value that identifies the root vertex
       (table adj relation _id _edge _void ...)         ;; maps src id -> edge (tuple of dest id, edge weight)
       (table pi function _id _id _void ())             ;; (output): maps each node to its parent in the mst
       (table cost->nodes relation _wt _id _void ())    ;; maps each cost to the nodes reachable with that cost
       (table node->cost function _id _wt _void ())     ;; maps each node to the best cost found so far
       (table q function _id _void _void ()))           ;; q: lists vertices whose succs haven't been analyzed)
     (kernels
       (kernel init-node-cost
         (wait)
         (on-entry)
         (read
           (bind adj adj forall preserve ())
           (bind r r !exists preserve ((= (path _id) (adj i (path _id)))))) ;; exclude root vertex
         (body
           (let ((srcId (table-index adj)))
             ;; add to queue
             (cwrite 1 q srcId 0 0)
             ;; set best known cost to infinity
             (cwrite 1 node->cost srcId 32767 0)))
         (on-success)
         (on-failure))
       ;; initialize data structures for root vertex
       (kernel init-root
         (wait)
         (on-entry)
         (read
           (bind r r exists preserve ()))
         (body
           (let ((rootId (table-index r)))
             (cwrite 1 q rootId 0 0)             ;; add to queue
             (cwrite 1 pi rootId 32767 0)        ;; set parent to unknown
             (cwrite 1 cost->nodes 0 0 0)        ;; note we can get to root at zero cost
             (cwrite 1 node->cost 0 0 0)))
         (on-success)
         (on-failure))
       ;; process succs of minimum-cost node
       (kernel process-succs
         (wait)
         (on-entry)
         (read
           (bind cost->nodes cost->nodes exists delete ())                          ;; select minimum cost node
           (bind q1 q exists delete ((= (path _id) (cost->nodes e (path _id)))))    ;; dequeue it
           (bind adj adj forall preserve ((= (path _id) (q1 i (path _id)))))        ;; find successor edges
           (bind node->cost node->cost exists preserve                              ;; find succ cost
             ((= (path _id) (adj e (path _edge dest)))))
           ;; the following is ok because we know that it will never match the row deleted from 'q'
           ;; above (provided that nodes are not immediate successors of themselves...)
           (bind q2 q exists preserve ((= (path _id) (node->cost i (path _id))))))  ;; make sure succ in q
         (body
           (let* ((adj-weight (project (path _edge wt) (table-ephemeral adj)))
                  (adj-dest (project (path _edge dest) (table-ephemeral adj)))
                  (adj-src (table-index adj))
                  (vertex-weight (table-index cost->nodes))
                  (best-weight (table-ephemeral node->cost)))
             (let* ((better (< adj-weight best-weight))
                    (not-better (lnot better)))
               ;; add succ case
               (cwrite better pi adj-dest adj-src 0)
               (cwrite better cost->nodes adj-weight adj-dest 0)
               (cwrite better node->cost  adj-dest adj-weight 0))))
         (on-success)
         (on-failure)))
     (schedule
       (begin init-node-cost init-root (while process-succs)))))
```

Figure 33: Minimum spanning tree (graph is the example from [CLR90], p. 508)

| mst | init | step | bkt-init | read | write | delete | succeed | fail | signal |
|---|---|---|---|---|---|---|---|---|---|
| program-root-203 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| init-node-cost | 1 | 27 | 11 | 30 | 52 | 0 | 1 | 0 | 1 |
| init-root | 1 | 2 | 1 | 1 | 4 | 0 | 1 | 0 | 1 |
| process-succs | 8 | 22 | 103 | 136 | 36 | 28 | 7 | 1 | 7 |
| total | 10 | 52 | 115 | 167 | 92 | 28 | 9 | 2 | 10 |

Figure 34: Minimum spanning tree statistics

### 8.1.6 Minimum spanning tree

The minimum spanning tree computation is a sequential dynamic programming algorithm. The outer loop dequeues the least-cost element from a priority queue, and processes all successors, updating the cost table when an improved path is found. Thus, only the inner loop (examining successors) can be streamed. One interesting feature of this example is the use of conditional writes: the cost relation and parent-of functions are updated only if an improved edge is found. Another is the need to represent the cost relation as a pair of functions `cost->node` and `node->cost` because the prototype supports only indexing, not full relational searching. The statistics for a small graph (obtained from [CLR90]) show that the use of the ephemeral field in the adjacency table improves locality: both kernels that access this table to obtain successor information achieve more than one memory read per table "bucket" loaded.

### 8.1.7 Convolution

We implemented two versions of one-dimensional convolution:

- `convolve-shift-reduce` (Fig. 35): This implementation is essentially an expansion phase followed by a reduction phase—not much different from matrix multiplication. The expansion kernel walks the input vector once, multiplying each element by all of the kernel values, and writing the results to appropriately shifted positions in an intermediate table. The reduction kernel walks the intermediate table on the "output position" axis, and sums the pre-multiplied values on the "sliding window" axis.

- `convolve-tiled` (Figs. 36 and 37): This is basically a 4-tiled version of `convolve-shift-reduce` that not only performs multiplication, but also computes all partial sums that can be computed given one tile's worth of input. This data is written to an intermediate structure (four distinct tables are needed because we are only allowed one real-valued column—see Section6.6 per table). The second pass then combines the partial results from tile $i$ with the input from tile $i + 1$, yielding the final answer. This code assumes that the input can be divided evenly into tiles—otherwise, we would need three additional "cleanup" kernels to handle the triple, double, and single "leftover input" cases.

  This strategy depends crucially on the fact that each output value depends (transitively) on only a fixed input range; fully transitive dependences (e.g., prefix scan) require additional effort. A different way of understanding this algorithm is to view it as a tiled, degenerate instance of the tree-based prefix scan algorithm in [Ble90], in which the left-to-right propagation of values is limited to adjacent tiles by restricting the maximal tree height to 2. Yet another way to view this code is as an oddly-stratified version of software pipelining, in which pipelining (execution of different iterations of multiple source-level loop instances at once) takes place only within tiles, with a form of "spilling" propagating dependences to memory, whose use must be delayed until the next activation of the streaming kernel.

As we can see in Figure 38, both implementations perform the same number of reads in the first pass, but the tiled version performs only 1/4 as many writes— and 1/4 of those writes contribute directly to the result, resulting in fewer writes in the second pass as well. We could make this even more efficient if we could distinguish between tile indices (stored in the index column) and element offsets (stored in the ephemeral column), but this is not possible because the prototype does not allow us to constrain ephemeral values.

```
(program convolve-shift-reduce
  (typedefs
    (typedef _i     (int 8))
    (typedef _V     real)
    (typedef _void  (int 0)))
  (tables
    (table x function _i _void _v ...)
    (table s0 function _i _void _v ())
    (table s1 function _i _void _v ())
    (table s2 function _i _void _v ())
    (table s3 function _i _void _v ())
    (table y function _i _void _v ()))
  (kernels
    (kernel shift-and-multiply
      (wait)
      (on-entry)
      (read
        (bind x x forall preserve ()))
      (body
        (let ((k0 0.0625) (k1 0.125) (k2 0.25) (k3 0.50))
          (cwrite 1 s0 (table-index x) 0 (* (table-data x) k0))
          (cwrite 1 s1 (- (table-index x) 1) 0 (* (table-data x) k1))
          (cwrite 1 s2 (- (table-index x) 2) 0 (* (table-data x) k2))
          (cwrite 1 s3 (- (table-index x) 3) 0 (* (table-data x) k3))))
      (on-success)
      (on-failure))
    (kernel add
      (wait)
      (on-entry)
      (read
        (bind s0 s0 forall preserve ())
        (bind s1 s1 exists preserve ((= (path _i) (s0 i (path _i)))))
        (bind s2 s2 exists preserve ((= (path _i) (s1 i (path _i)))))
        (bind s3 s3 exists preserve ((= (path _i) (s2 i (path _i))))))
      (body
        (let ((sum (+ (+ (table-data s0) (table-data s1))
                      (+ (table-data s2) (table-data s3)))))
          (cwrite 1 y (table-index s0) 0 sum)))
      (on-success)
      (on-failure)))
  (schedule (begin shift-and-multiply add))))
```

Figure 35: Convolution, shift-and-reduce (kernel length= 4, input length = 128)

```
fine convolve-tiled
(program convolve-tiled
  (typedefs
    (typedef _i     (int 8))
    (typedef _V     real)
    (typedef _void  (int 0)))
  (tables
    (table x function _i _void _v ...)
    (table i123 function _i _void _v ())
    (table i23  function _i _void _v ())
    (table i3   function _i _void _v ())
    (table y function _i _void _v ()))
  (kernels
    (kernel pass1
      (wait)
      (on-entry)
      (read
        (bind s0 x forall preserve ((= 1:0 0)))
        (bind s1 x exists preserve ((= 1:0 1) (= 7:2 (s0 i 7:2))))
        (bind s2 x exists preserve ((= 1:0 2) (= 7:2 (s1 i 7:2))))
        (bind s3 x exists preserve ((= 1:0 3) (= 7:2 (s2 i 7:2)))))
      (body
        (let ((k0 0.0625) (k1 0.125) (k2 0.25) (k3 0.50))
          (let ((p0123 (+ (* k0 (table-data s0))
                          (+ (* k1 (table-data s1))
                            (+ (* k2 (table-data s2))
                              (* k3 (table-data s3))))))
                (p123 (+ (* k0 (table-data s1))
                         (+ (* k1 (table-data s2))
                           (* k2 (table-data s3)))))
                (p23 (+ (* k0 (table-data s2))
                        (* k1 (table-data s3))))
                (p3 (* k0 (table-data s3)))
                (index (+ (shr (table-index s0) 2) 1)))
            (begin
              ;; write partial results into i**
              ;; offset them by 1 so they'll match proper blocks in pass1
              (cwrite 1 y (table-index s0) 0 p0123)
              (cwrite 1 i123 index 0 p123)
              (cwrite 1 i23 index 0 p23)
              (cwrite 1 i3 index 0 p3)))))
      (on-success)
      (on-failure))
  (schedule (begin pass1 pass2)))
```

Figure 36: Convolution, tiled (kernel length= 4, input length = 128), pass 1

```
  ;; given partial results, compute results for each remaining 4-block
  (kernel pass2
    (wait)
    (on-entry)
    (read
      (bind i123 i123 forall preserve ())
      (bind i23 i23 exists preserve ((= 7:0 (i123 i 7:0))))
      (bind i3 i3 exists preserve ((= 7:0 (i23 i 7:0))))
      (bind s0 x exists preserve ((= 1:0 0) (= 7:2 (i3 i 5:0))))
      (bind s1 x exists preserve ((= 1:0 1) (= 7:2 (s0 i 7:2))))
      (bind s2 x exists preserve ((= 1:0 2) (= 7:2 (s1 i 7:2))))
      (bind s3 x exists preserve ((= 1:0 3) (= 7:2 (s2 i 7:2)))))
    (body
      (let ((k0 0.0625) (k1 0.125) (k2 0.25) (k3 0.50))
        (let ((r0 (+ (table-data i123)
                     (* k3 (table-data s0))))
              (r1 (+ (table-data i23)
                     (+ (* k2 (table-data s0))
                       (* k3 (table-data s1)))))
              (r2 (+ (table-data i3)
                     (+ (* k1 (table-data s0))
                       (+ (* k2 (table-data s1))
                         (* k3 (table-data s2))))))
              (index (- (table-index s0) 3)))
          (cwrite 1 y index 0 r0)
          (cwrite 1 y (+ index 1) 0 r1)
          (cwrite 1 y (+ index 2) 0 r2))))
    (on-success)
    (on-failure)))
(schedule (begin pass1 pass2)))
```

Figure 37: Convolution, tiled (kernel length= 4, input length = 128), pass 2

| convolve-shift-reduce | init | step | bkt-init | read | write | delete | succeed | fail | signal |
|---|---|---|---|---|---|---|---|---|---|
| program-root-206 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| shift-and-multiply | 1 | 129 | 128 | 128 | 512 | 0 | 1 | 0 | 1 |
| add | 1 | 126 | 506 | 506 | 125 | 0 | 1 | 0 | 0 |
| total | 2 | 256 | 634 | 634 | 637 | 0 | 2 | 1 | 2 |
| convolve-tiled | init | step | bkt-init | read | write | delete | succeed | fail | signal |
| program-root-212 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| pass1 | 1 | 33 | 128 | 128 | 128 | 0 | 1 | 0 | 1 |
| pass2 | 1 | 32 | 220 | 220 | 93 | 0 | 1 | 0 | 0 |
| total | 2 | 66 | 348 | 348 | 221 | 0 | 2 | 1 | 2 |

Figure 38: Convolution statistics (128 elements)

```
(program prefix-scan-iterative
  (typedefs
    (typedef _tag (int 8))
    (typedef _void (int 0)))
  (tables
    (table x function _tag _void real ...)
    (table y function _tag _void real ())
    (table acc function _tag _void real ((0 0 0.0))))
  (kernels
    (kernel scan-loop
      (wait)
      (on-entry)
      (read
        (bind x x exists delete ())
        (bind acc acc exists delete ()))
      (body
        (let ((sum (+ (table-data x)
                      (table-data acc))))
          (cwrite 1 y (table-index x) 0 sum)
          (cwrite 1 acc 0 0 sum)))
      (on-success)
      (on-failure)))
  (schedule (while scan-loop)))
```

Figure 39: Iterative prefix scan

```
(program prefix-scan-iterative-by4
  (typedefs
    (typedef _tag (int 8))
    (typedef _void (int 0)))
  (tables
    (table x function _tag _void real ,(ex:make-dense-vector-init-bcs 0 256 1))
    (table y function _tag _void real ())
    (table acc function _tag _void real ((0 0 0.0))))
  (kernels
    (kernel scan-loop
      (wait)
      (on-entry)
      ;; relies on increasing-order processing of indices
      ;; (otherwise, we'd need an explicit index to drive the loop)
      (read
        (bind x0 x exists delete ((= 1:0 0)))
        (bind x1 x exists delete ((= 1:0 1) (= 7:2 (x0 i 7:2))))
        (bind x2 x exists delete ((= 1:0 2) (= 7:2 (x1 i 7:2))))
        (bind x3 x exists delete ((= 1:0 3) (= 7:2 (x2 i 7:2))))
        (bind acc acc exists delete ()))
      (body
        (let* ((r0 (+ (table-data acc) (table-data x0)))
               (r1 (+ r0 (table-data x1)))
               (r2 (+ r1 (table-data x2)))
               (r3 (+ r2 (table-data x3))))
          (cwrite 1 y (table-index x0) 0 r0)
          (cwrite 1 y (table-index x1) 0 r1)
          (cwrite 1 y (table-index x2) 0 r2)
          (cwrite 1 y (table-index x3) 0 r3)
          (cwrite 1 acc 0 0 r3)))
      (on-success)
      (on-failure)))
  (schedule (while scan-loop)))
```

Figure 40: Iterative prefix scan

```
(program prefix-scan-tree-by4
  (typedefs
    (typedef _l (int 8))
    (typedef _i (int 16))
    (typedef _v real)
    (typedef _void (int 0))
    (typedef _li (struct (index _i) (level _l))))
  (tables
    (table up function _li _void real
      ,(let ((input-init (ex:make-dense-vector-init-bcs 0 256 1)))
         (map (lambda (init)
                (cons (list (car init) 1) (cdr init)))
           input-init)))
    (table down function _li _void real ())
    (table level function _l _void _void ((1 0 0))))
  (kernels
    (kernel scan-up
      (wait)
      (on-entry)
      (read
        (bind level level exists preserve ())
        ;; join level field, restrict index mod4 = 0
        (bind u0 up forall preserve ((= 23:16 (level i 7:0)) (= 1:0 0)))
        ;; join level and index fields, except require index mod4 = 1
        (bind u1 up exists preserve ((= 23:2 (u0 i 23:2)) (= 1:0 1)))
        ;; join level and index fields, except require index mod4 = 2
        (bind u2 up exists preserve ((= 23:2 (u1 i 23:2)) (= 1:0 2)))
        ;; join level and index fields, except require index mod4 = 3
        (bind u3 up exists preserve ((= 23:2 (u2 i 23:2)) (= 1:0 3))))
      (body
        (let ((sum (+ (+ (table-data u0) (table-data u1))
                      (+ (table-data u2) (table-data u3))))
              (nlevel (+ (table-index level) 1))
              (nindex (project 15:2 (table-index u0))))
          (cwrite 1 up (inject nindex 23:16 nlevel) 0 sum)))
      (on-success)
      (on-failure))
    (kernel bump-level
      (wait)
      (on-entry)
      (read
        (bind level level exists delete ()))
      (body
        (cwrite 1 level (+ (table-index level) 1) 0 0))
      (on-success)
      (on-failure))
    (kernel init-down
      (wait)
      (on-entry)
      (read
        (bind level level exists delete ()))
      (body
        (begin
          (cwrite 1 level (- (table-index level) 1) 0 0)
          (cwrite 1 down (inject 0 23:16 (- (table-index level) 1)) 0 0.0)
          ;;(cwrite 1 child-level (- (table-index level) 1) 0 0)
          ))
      (on-success)
      (on-failure))
```

Figure 41: Tree prefix scan, upward pass

```
  (kernel scan-down
    (wait)
    (on-entry)
    (read
      (bind level level exists preserve ())
      ;; join level fields
      (bind down down forall delete  ((= 23:16 (level i 7:0))))
      (bind u0 up exists delete ((= 23:16 (down i 23:16))
                                 (= 15:2 (down i 13:0))
                                 (= 1:0 0)))
      ;; join level and index fields, except require index mod4 = 1
      (bind u1 up exists delete ((= 23:2 (u0 i 23:2)) (= 1:0 1)))
      ;; join level and index fields, except require index mod4 = 2
      (bind u2 up exists delete ((= 23:2 (u1 i 23:2)) (= 1:0 2)))
      ;; join level and index fields, except require index mod4 = 3
      (bind u3 up exists delete ((= 23:2 (u2 i 23:2)) (= 1:0 3))))
    (body
      (let* ((r0 (table-data down))
             (r1 (+ r0 (table-data u0)))
             (r2 (+ r1 (table-data u1)))
             (r3 (+ r2 (table-data u2)))
             (index (shl (project 15:0 (table-index down)) 2)))
        (begin
          (let ((child-level (- (table-index level) 1)))
            (cwrite 1 down (inject index 23:16 child-level) 0 r0)
            (cwrite 1 down (inject (+ index 1) 23:16 child-level) 0 r1)
            (cwrite 1 down (inject (+ index 2) 23:16 child-level) 0 r2)
            (cwrite 1 down (inject (+ index 3) 23:16 child-level) 0 r3)))))
    (on-success)
    (on-failure))
  (kernel dec-level
    (wait)
    (on-entry)
    (read
      (bind level level exists delete ()))
    (body
      (cwrite (> (table-index level) 0) level (- (table-index level) 1) 0 0))
    (on-success)
    (on-failure)))
(schedule (begin
            (while scan-up bump-level)
            init-down
            (while scan-down dec-level))))
```

Figure 42: Tree prefix scan, downward pass

| prefix-scan-iterative | init | step | bkt-init | read | write | delete | succeed | fail | signal |
|---|---|---|---|---|---|---|---|---|---|
| program-root-218 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| scan-loop | 257 | 513 | 512 | 512 | 512 | 512 | 256 | 1 | 256 |
| total | 257 | 514 | 512 | 512 | 512 | 512 | 256 | 2 | 257 |
| prefix-scan-iterative-by4 | init | step | bkt-init | read | write | delete | succeed | fail | signal |
| program-root-224 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| scan-loop | 65 | 129 | 320 | 320 | 320 | 320 | 64 | 1 | 64 |
| total | 65 | 130 | 320 | 320 | 320 | 320 | 64 | 2 | 65 |
| prefix-scan-tree-by4 | init | step | bkt-init | read | write | delete | succeed | fail | signal |
| program-root-182 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| scan-up | 5 | 90 | 346 | 346 | 85 | 0 | 4 | 1 | 5 |
| bump-level | 4 | 8 | 4 | 4 | 4 | 4 | 4 | 0 | 4 |
| init-down | 1 | 2 | 1 | 1 | 2 | 1 | 1 | 0 | 1 |
| scan-down | 5 | 90 | 686 | 686 | 340 | 425 | 4 | 1 | 4 |
| dec-level | 4 | 8 | 4 | 4 | 4 | 4 | 4 | 0 | 4 |
| total | 19 | 199 | 1041 | 1041 | 435 | 434 | 17 | 3 | 19 |

Figure 43: Prefix scan statistics (256 elements)

#### 8.1.8 Prefix scan

We present three algorithms for prefix scan over scalar addition:

- `prefix-scan-iterative` (Fig. 39): This is the obvious iterative solution that carries an accumulator, writing the current value at each step. We rely on the fact that index column values are enumerated in ascending order.
- `prefix-scan-iterative-by4` (Fig. 40): This is a 4-wide tiling of the iterative version. Relative to the non-tiled iterative version, it performs the same number of input reads, but 1/4 as many kernel initializations, accumulator reads, and accumulator writes.
- `prefix-scan-tree`

  (Figs. 41 and 42): This is a radix-4 adaptation of Blelloch's parallel prefix scan [Ble90]. The algorithm operates in two passes. The first pass operates exactly like a tree-based reduction scheme, except that it retains all of the intermediate sums. Thus, we add a bitrange representing the current "level" in the reduction tree to the data representation; the first ("upward") pass reads from level $n$ and writes to level $n+1$. In the second ("downward") pass, each tile propagates its value to the leftmost child of its next-rightmost sibling in the tree.

Figure 43 summarizes the processing of a 64-element input vector. In terms of memory traffic, the tiled iterative version does best; the tree-based version performs the fewest kernel initializations. The determination of the best performer depends on the relative costs of streaming memory accesses, non-streaming memory accesses, and kernel initialization. In the face of this uncertainty, the best argument in favor of the tree-based algorithm is that it supports parallelism. Unlike the vector reduction case, where the results of disjoint input subranges (perhaps computed by tiled iterative methods) can easily be combined, prefix scan requires the propagation of sums at all levels, as in the tree-based algorithm. In terms of interpreter "steps" (a coarse measure that does not account for streaming effects) the tree-based algorithm beats the tiled iterative version when four or more kernels can be executed in parallel.

#### 8.1.9 Regular expression recognition

Recognition of regular expression recognition involves the modeling of a deterministic finite-state machine. One approach is essentially interpretive in nature, sequentially modeling state changes as each input value

```
(program fsm-iterative
  (typedefs
    (typedef _int (int 8))
    (typedef _tag (int 8))
    (typedef _state (int 8))
    (typedef _input (int 8))
    (typedef _transition (struct (s _state) (i _input) (ns _state)))
    (typedef _void (int 0)))
  (tables
    (table transitions function _transition _void _void
      (((0 0 1) 0 0)   ;; state 0, input 0 --> new state 0
       ((0 1 0) 0 0)   ;; state 0, input 1 --> new state 0
       ((1 0 1) 0 0)   ;; etc.
       ((1 1 2) 0 0)
       ((2 0 1) 0 0)
       ((2 1 3) 0 0)
       ((3 0 1) 0 0)
       ((3 1 0) 0 0)
       ((3 2 4) 0 0)))
    (table input function _tag _input _void ...)  ;; 35 0s followed by 1 1 0 1 1 2
    (table success function _int _void _void ()) ;; nonempty if recognizer succeeded
    (table state function _int _void _void ((0 0 0))))  ;; assume start state is 0
  (kernels
    (kernel step
      (wait)
      (on-entry)
      (read
        ;; fetch current state
        (bind state state exists delete ())
        ;; fetch current input
        ;; (index contains only tags, so tuples are read in tag order)
        (bind input input exists delete ())
        ;; given state and input, find matching transition (and new state)
        (bind map transitions exists preserve ((= (path _transition i) (input e (path _input)))
                                               (= (path _transition s) (state i (path _state))))))
      (body
        (cwrite 1 state (project (path _transition ns) (table-index map)) 0 0))
      (on-success)
      (on-failure))
    (kernel eval-success
      (wait)
      (on-entry)
      (read
        ;; make sure no input is left
        (bind input input !exists preserve ())
        ;; make sure we're in end state (should we parameterize this?)
        (bind state state exists preserve ((= (path _state) 4))))
      (body
        ;; if we got here write a tuple to indicate recognition
        (cwrite 1 success 0 0 0))
      (on-success)
      (on-failure)))
  (schedule (begin (while step) eval-success))))
```

Figure 44: Iterative FSM recognizer

```
(define fsm-iterative-by4
  `(program fsm-iterative-by4
     (typedefs ...)                                          ;; same as fsm-iterative
     (tables
       (table transitions function _transition _void _void ...) ;; see fsm-iterative
       (table input function _tag _input _void ...)            ;; see fsm-iterative

       (table index function _int _void _void ((0 0 0)))
       (table success function _int _void _void ()) ;; nonempty if recognizer succeeded
       (table state function _int _void _void ((0 0 0))))  ;; assume start state is 0
     (kernels
       (kernel step-by-4
         (wait)
         (on-entry)
         (read
           ;; fetch current index
           (bind index index exists delete ())
           ;; fetch current state
           (bind state state exists delete ())
           ;; fetch current input (using index)
           (bind input0 input exists delete ((= 7:2 (index i 5:0))
                                             (= 1:0 0)))
           (bind input1 input exists delete ((= 7:2 (index i 5:0))
                                             (= 1:0 1)))
           (bind input2 input exists delete ((= 7:2 (index i 5:0))
                                             (= 1:0 2)))
           (bind input3 input exists delete ((= 7:2 (index i 5:0))
                                             (= 1:0 3)))
           ;; find a transition accepting (state, input0)
           (bind map0 transitions exists preserve ((= (path _transition i) (input0 e (path _input)))
                                                   (= (path _transition s) (state i (path _state)))))
           ;; find a transition accepting (new state, input1)
           (bind map1 transitions exists preserve ((= (path _transition i) (input1 e (path _input)))
                                                   (= (path _transition s) (map0 i (path _transition ns)))))
           ;; find a transition accepting (new state, input2)
           (bind map2 transitions exists preserve ((= (path _transition i) (input2 e (path _input)))
                                                   (= (path _transition s) (map1 i (path _transition ns)))))
           ;; find a transition accepting (new state, input3)
           (bind map3 transitions exists preserve ((= (path _transition i) (input3 e (path _input)))
                                                   (= (path _transition s) (map2 i (path _transition ns))))))
         (body
           (begin
             (cwrite 1 state (project (path _transition ns) (table-index map3)) 0 0)
             (cwrite 1 index (+ (table-index index) 1) 0 0)))
         (On-success)
         (on-failure))
       (kernel cleanup
         (wait)
         (on-entry)
         (read
           ;; fetch current state
           (bind state state exists delete ())
           ;; fetch current input
           (bind input input exists delete ())
           ;; given state and input, find matching transition (and new state)
           (bind map transitions exists preserve ((= (path _transition i) (input e (path _input)))
                                                  (= (path _transition s) (state i (path _state))))))
         (body
           (cwrite 1 state (project (path _transition ns) (table-index map)) 0 0))
         (on-success)
         (on-failure))
       (kernel eval-success
         (wait)
         (on-entry)
         (read
           ;; make sure no input is left
           (bind input input !exists preserve ())
           ;; make sure we're in end state (should we parameterize this?)
           (bind state state exists preserve ((= (path _state) 4))))
         (body
           ;; if we got here write a tuple to indicate recognition
           (cwrite 1 success 0 0 0))
         (on-success)
         (on-failure)))
     (schedule (begin (while step-by-4) (while cleanup) eval-success))))
```

Figure 45: Iterative FSM recognizer, tiled x4

```
(program fsm-tree-by4
  (typedefs
    (typedef _int (int 8))
    (typedef _tag (int 8))
    (typedef _state (int 8))
    (typedef _input (int 8))
    ;; cols are input state, input value, new state
    (typedef _transition (struct (s _state) (i _input) (ns _state)))
    ;; cols are level, index, rightmost input pos represented, input state, output state
    ;; tag is in least-significant bits
    (typedef _map (struct  (t _tag) (rt _tag) (l _int) (s _state) (ns _state)))
    (typedef _void (int 0)))
  (tables
    (table transitions function _transition _void _void ...)    ;; see fsm-iterative
    (table final-state-id function _int _void _void ((4 0 0)))  ;; see fsm-iterative
    (table input function _tag _input _void ...)
    (table level function _int _void _void ((1 0 0)))
    (table input-max-index function _int _void _void ((42 0 0))) ;; 42 for long example
    (table maps function _map _void _void ())
    (table done function _int _void _void ())
    (table success function _void _void _void ()))
  (kernels
    (kernel input-to-maps-by4
      (wait)
      (on-entry)
      (read
        ;; fetch input
        (bind input0 input forall delete ((= 1:0 0)))
        (bind input1 input exists delete ((= 7:2 (input0 i 7:2)) (= 1:0 1)))
        (bind input2 input exists delete ((= 7:2 (input0 i 7:2)) (= 1:0 2)))
        (bind input3 input exists delete ((= 7:2 (input0 i 7:2)) (= 1:0 3)))
        ;; fetch transitions consuming input
        (bind transitions0 transitions forall preserve
          ((= (path _transition i) (input0 e (path _input)))))
        ;; and so on...
        (bind transitions1 transitions forall preserve
          ((= (path _transition i) (input1 e (path _input)))
           (= (path _transition s) (transitions0 i (path _transition ns)))))
        (bind transitions2 transitions forall preserve
          ((= (path _transition i) (input2 e (path _input)))
           (= (path _transition s) (transitions1 i (path _transition ns)))))
        (bind transitions3 transitions forall preserve
          ((= (path _transition i) (input3 e (path _input)))
           (= (path _transition s) (transitions2 i (path _transition ns))))))
      (body
        (let* ((nlevel 1)
               (tag (table-index input0))
               (rtag (table-index input3))
               (initial-state (project (path _map s) (table-index transitions0)))
               (final-state (project (path _map ns) (table-index transitions3))))
          (let* ((nmap (shr tag 2))
                 (nmap (inject nmap (path _map rt) rtag))
                 (nmap (inject nmap (path _map l) nlevel))
                 (nmap (inject nmap (path _map s) initial-state))
                 (nmap (inject nmap (path _map ns) final-state)))
            (cwrite 1 maps nmap 0 0))))
      (on-success)
      (on-failure))
```

Figure 46: Tree (function composition) FSM recognizer, tiled x4, part 1

```
(kernel input-to-maps-by1
  (wait)
  (on-entry)
  (read
    ;; fetch input, in order
    (bind input input forall preserve ())
    ;; fetch transitions consuming input
    (bind transitions transitions forall preserve ((= (path _transition i) (input e (path _input))))))
  (body
    (let ((transition (table-index transitions))
          (tag (table-index input)))
      (let ((state (project (path _transition s) transition))
            (nstate (project (path _transition ns) transition))
            (ntag (+ (shr tag 2) (project 1:0 tag))))
        (let* ((nmap ntag)
               (nmap (inject nmap (path _map rt) tag))
               (nmap (inject nmap (path _map l) 1))
               (nmap (inject nmap (path _map s) state))
               (nmap (inject nmap (path _map ns) nstate)))
          (cwrite 1 maps nmap 0 0)))))
  (on-success)
  (on-failure))
(kernel compose-by4
  (wait)
  (on-entry)
  (read
    (bind level level exists preserve ())
    (bind m0 maps forall delete ((= 1:0 0)
                                 (= (path _map l) (level i (path _int)))))
    (bind m1 maps forall preserve ((= 1:0 1)
                                   (= 7:2 (m0 i 7:2))
                                   (= (path _map s) (m0 i (path _map ns)))
                                   (= (path _map l) (m0 i (path _map l)))))
    (bind m2 maps forall preserve ((= 1:0 2)
                                   (= 7:2 (m1 i 7:2))
                                   (= (path _map s) (m1 i (path _map ns)))
                                   (= (path _map l) (m1 i (path _map l)))))
    (bind m3 maps forall preserve ((= 1:0 3)
                                   (= 7:2 (m2 i 7:2))
                                   (= (path _map s) (m2 i (path _map ns)))
                                   (= (path _map l) (m2 i (path _map l))))))
  (body
    (let* ((nlevel (+ 1 (table-index level)))
           (tag (project (path _map t) (table-index m0)))
           (rtag (project (path _map rt) (table-index m3)))
           (initial-state (project (path _map s) (table-index m0)))
           (final-state (project (path _map ns) (table-index m3))))
      (let* ((nmap (shr tag 2))
             (nmap (inject nmap (path _map rt) rtag))
             (nmap (inject nmap (path _map l) nlevel))
             (nmap (inject nmap (path _map s) initial-state))
             (nmap (inject nmap (path _map ns) final-state)))
        (cwrite 1 maps nmap 0 0))))
  (on-success)
  (on-failure))
```

Figure 47: Tree (function composition) FSM recognizer, tiled x4, part 2

```
(kernel compose-by3
  (wait)
  (on-entry)
  (read
    (bind level level exists preserve ())
    (bind m0 maps forall delete ((= 1:0 0)
                                 (= (path _map l) (level i (path _int)))))
    (bind m1 maps forall preserve ((= 1:0 1)
                                   (= 7:2 (m0 i 7:2))
                                   (= (path _map s) (m0 i (path _map ns)))
                                   (= (path _map l) (m0 i (path _map l)))))
    (bind m2 maps forall preserve ((= 1:0 2)
                                   (= 7:2 (m1 i 7:2))
                                   (= (path _map s) (m1 i (path _map ns)))
                                   (= (path _map l) (m1 i (path _map l))))))
  (body
    (let* ((nlevel (+ 1 (table-index level)))
           (tag (project (path _map t) (table-index m0)))
           (rtag (project (path _map rt) (table-index m2)))
           (initial-state (project (path _map s) (table-index m0)))
           (final-state (project (path _map ns) (table-index m2))))
      (let* ((nmap (shr tag 2))
             (nmap (inject nmap (path _map rt) rtag))
             (nmap (inject nmap (path _map l) nlevel))
             (nmap (inject nmap (path _map s) initial-state))
             (nmap (inject nmap (path _map ns) final-state)))
        (cwrite 1 maps nmap 0 0))))
  (on-success)
  (on-failure))

(kernel compose-by2
  (wait)
  (on-entry)
  (read
    (bind level level exists preserve ())
    (bind m0 maps forall delete ((= 1:0 0)
                                 (= (path _map l) (level i (path _int)))))
    (bind m1 maps forall preserve ((= 1:0 1)
                                   (= 7:2 (m0 i 7:2))
                                   (= (path _map s) (m0 i (path _map ns)))
                                   (= (path _map l) (m0 i (path _map l))))))
  (body
    (let* ((nlevel (+ 1 (table-index level)))
           (tag (project (path _map t) (table-index m0)))
           (rtag (project (path _map rt) (table-index m1)))
           (initial-state (project (path _map s) (table-index m0)))
           (final-state (project (path _map ns) (table-index m1))))
      (let* ((nmap (shr tag 2))
             (nmap (inject nmap (path _map rt) rtag))
             (nmap (inject nmap (path _map l) nlevel))
             (nmap (inject nmap (path _map s) initial-state))
             (nmap (inject nmap (path _map ns) final-state)))
        (cwrite 1 maps nmap 0 0))))
  (on-success)
  (on-failure))
```

Figure 48: Tree (function composition) FSM recognizer, tiled x4, part 3

```
(kernel compose-by1 ;; just copies value
  (wait)
  (on-entry)
  (read
    (bind level level exists preserve ())
    (bind m0 maps forall delete ((= 1:0 0)
                                 (= (path _map l) (level i (path _int)))))
    (bind final-state-id final-state-id exists preserve ()))
  (body
    ;; if tag=0, signal the fact that we're done by writing to done
    ;; more efficient to use an explicit signal?  but how to integrate with schedule?
    (let* ((nlevel (+ 1 (table-index level)))
           (tag (project (path _map t) (table-index m0)))
           (rtag (project (path _map rt) (table-index m0)))
           (nmap (shr tag 2))
           (nmap (inject nmap (path _map rt) rtag))
           (nmap (inject nmap (path _map l) nlevel))
           (nmap (inject nmap (path _map s) 0))
           (nmap (inject nmap (path _map ns) (table-index final-state-id))))
      (cwrite (= tag 0) done 0 0 0)
      ;; write tag=0 case back out, to enable success check
      (cwrite (= tag 0) maps (table-index m0) (table-ephemeral m0) (table-data m0))
      (cwrite 1 maps nmap 0 0)))   ;; was (!= tag 0)
  (on-success)
  (on-failure))

(kernel bump-index
  (wait)
  (on-entry)
  (read
    (bind level level exists delete ()))
  (body
    (let* ((nlevel (+ 1 (table-index level))))
      (cwrite 1 level nlevel 0 0)))
  (on-success)
  (on-failure))
(kernel exit-loop-on-done
  (wait)
  (on-entry)
  (read
    (bind done done exists preserve ()))
  (body
    9999)
  (on-success)
  (on-failure))
```

Figure 49: Tree (function composition) FSM recognizer, tiled x4, part 4

```
  (kernel eval-success
    (wait)
    (on-entry)
    (read
      ;; grab end state
      (bind final-state-id final-state-id exists preserve ())
      ;; ensure map entry goes from 0 to end state
      (bind maps maps exists preserve ((= (path _map s) 0)
                                       (= (path _map ns) (final-state-id i (path _int)))))
      ;; read input length
      (bind input-max-index input-max-index exists preserve ()))
    (body
      ;; ensure map entry represents consumption of correct amt of input
      (cwrite
        (= (table-index input-max-index)
          (project (path _map rt) (table-index maps)))
        success
        0
        0
        0))
    (on-success)
    (on-failure)))
(schedule (begin
            input-to-maps-by4
            input-to-maps-by1
            (until
              (begin
                compose-by4    ;; always do x4 tiles
                (or            ;; select cleanup function
                  compose-by3
                  compose-by2
                  compose-by1)
                bump-index)
              exit-loop-on-done)
            eval-success))))
```

Figure 50: Tree (function composition) FSM recognizer, tiled x4, part 5

| fsm-iterative | init | step | bkt-init | read | write | delete | succeed | fail | signal |
|---|---|---|---|---|---|---|---|---|---|
| program-root-756 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| step | 44 | 87 | 130 | 130 | 43 | 86 | 43 | 1 | 44 |
| eval-success | 1 | 2 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| total | 45 | 90 | 131 | 131 | 44 | 86 | 44 | 2 | 45 |
| fsm-iterative-by4 | init | step | bkt-init | read | write | delete | succeed | fail | signal |
| program-root-768 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| step-by-4 | 11 | 21 | 105 | 105 | 20 | 60 | 10 | 1 | 11 |
| cleanup | 4 | 7 | 10 | 10 | 3 | 6 | 3 | 1 | 4 |
| eval-success | 1 | 2 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| total | 16 | 31 | 116 | 116 | 24 | 66 | 14 | 3 | 16 |
| fsm-tree-by4 | init | step | bkt-init | read | write | delete | succeed | fail | signal |
| program-root-784 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| seq-succ-collector-776 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 2 | 2 |
| input-to-maps-by4 | 1 | 41 | 203 | 203 | 40 | 160 | 1 | 0 | 1 |
| input-to-maps-by1 | 1 | 10 | 12 | 12 | 9 | 0 | 1 | 0 | 1 |
| compose-by4 | 3 | 7 | 21 | 21 | 4 | 4 | 2 | 1 | 3 |
| compose-by3 | 3 | 3 | 5 | 5 | 0 | 0 | 0 | 3 | 3 |
| compose-by2 | 3 | 3 | 5 | 5 | 0 | 0 | 0 | 3 | 3 |
| compose-by1 | 3 | 5 | 7 | 7 | 4 | 2 | 2 | 1 | 3 |
| bump-index | 3 | 6 | 3 | 3 | 3 | 3 | 3 | 0 | 3 |
| exit-loop-on-done | 3 | 4 | 1 | 1 | 0 | 0 | 1 | 2 | 3 |
| eval-success | 1 | 2 | 3 | 3 | 1 | 0 | 1 | 0 | 0 |
| total | 21 | 84 | 260 | 260 | 61 | 169 | 11 | 13 | 23 |

Figure 51: Regular expression recognition statistics (41-element input)

is consumed. This can be tiled, using a more complex query to consume multiple inputs in a single kernel activation. A completely different approach relies on reformulating the problem as a map/reduce combination. The idea here is to compute the state-to-state transfer function for each input value, then reduce the function composition operator over the resultant vector of function representations. While far more complex than the iterative and tiled-iterative approaches, the function mapping / function composition approach is parallelizable.

We present three implementations:

- `fsm-iterative` (Fig. 44): This algorithm processes a single input per activation of the kernel `step`. The current input and current state form an index into the state transition function table; the resultant lookup produces a new state, which is used to update the current state value for the next iteration. After each iteration, we test to see if we have achieved recognition (i.e., the input has been consumed, and the current state is a terminal state).

- `fsm-iterative-by4` (Fig. 45): This algorithm is the obvious by-4 tiling of `fsm-iterative`. It processes four inputs at a time, updating the current state. To support inputs whose length is not a multiple of four, a "cleanup" kernel (identical to the kernel `step` in the by-1 iterative implementation) is invoked once the by-4 kernel can no longer process input.

- `fsm-tree-by4` (Figs. 46–50): This algorithm first computes the transfer function (represented by multiple rows in the table `maps` for each input), then composes the resultant functions. We attempt to offset the cost of the initial mapping phase (mapping from inputs to functions) by combining function construction with by-4 function composition. After this, a standard by-4 tree reduction performs the remainder of the function composition. Because we wish to allow inputs whose length is not a power of the tile size (in this case, 4), each pass of the tree reduction invokes one of three "cleanup" kernels (`compose-by3`, `compose-by2`, or `compose-by1`.

```
(program cfl-recognizer
  (typedefs
    (typedef _void (int 0))
    (typedef _int (int 8))
    (typedef _tag (int 8))                                  ;; input index
    (typedef _id (int 8))                                   ;; production id
    (typedef _lexeme (int 8))                               ;; lexeme values
    (typedef _tp (struct (id _id) (lexeme _lexeme)))        ;; unary (terminal) production
    (typedef _bp (struct (id _id) (rhs0 _id) (rhs1 _id)))   ;; binary (2 nonterminals) production
    (typedef _map (struct (id _id) (lo _tag) (hi _tag))))   ;; production instance
  (tables
    (table terminal-productions function _tp _void _void
      (((0 2) 0 0)
       ((1 0) 0 0)
       ((3 1) 0 0)))
    (table binary-productions function _bp _void _void
      (((0 1 2) 0 0)
       ((2 0 1) 0 0)
       ((0 3 4) 0 0)
       ((4 0 3) 0 0)))
    (table input function _tag _lexeme _void ...)           ;; 23-element dense tagged vector
    (table input-count function _int _void _void ((23 0 0)))
    (table maps function _map _void _void ())
    (table success function _int _void _void ()))
  (kernels
    ;; initialize maps from terminal productions
    (kernel init-maps
      (wait)
      (on-entry)
      (read
        (bind input input forall preserve ())
        (bind tp terminal-productions forall preserve ((= (path _tp lexeme) (input e (path _lexeme))))))
      (body
        (let ((tag (table-index input))
              (production (table-index tp)))
          (let* ((map (inject 0 (path _map id) (project (path _tp id) production)))
                 (map (inject map (path _map lo) tag))
                 (map (inject map (path _map hi) (+ tag 1))))   ;; was + 1
            (cwrite 1 maps map 0 0))))
      (on-success)
      (on-failure))
```

Figure 52: Context free language recognizer (chart parser), initialization

Figure 51 summarizes performance data for the recognition of a simple pattern on a 41-element input. Both the tiled iterative and tree-based algorithms perform the same number of by-4 compositions, and both must execute cleanup kernels due to the non-power-of-4 input length, but the overhead of converting inputs to functions forms the bulk of the cost in the function-composition case. As with prefix scan, the more complex tree-based algorithm provides a benefit only when parallelized x4 or higher.

It is also worth noting that these algorithms perform only recognition, and do not support the performance of any actions (e.g., generation of output) on state transitions. Should this be required, the iterative algorithms could be adapted fairly easily (e.g., writing to an output table as each input is consumed), while the tree-based algorithm would become significantly more complex.

### 8.1.10 Context-free expression recognition

This example is a naive, bottom-up chart parser. It requires that the language be described by a grammar in Chomsky Normal Form (CNF). That is, each production's right-hand side must be either a terminal a

```
  (kernel match
    (wait)
    (on-entry)
    (read
      ;; choose a production
      (bind prod binary-productions forall preserve ())
      ;; choose a lhs covered by production's left input
      (bind lhs maps forall preserve ((= (path _map id) (prod i (path _bp rhs0)))))
      ;; choose an adjacent rhs covered by production's right input
      (bind rhs maps exists preserve ((= (path _map lo) (lhs i (path _map hi)))
                                      (= (path _map id) (prod i (path _bp rhs1)))))
      ;; ensure new prod doesn't already exist
      (bind dup maps !exists preserve ((= (path _map id) (prod i (path _bp id)))
                                       (= (path _map lo) (lhs i (path _map lo)))
                                       (= (path _map hi) (rhs i (path _map hi))))))
    (body
      (let* ((id (project (path _map id) (table-index prod)))
             (lo (project (path _map lo) (table-index lhs)))
             (hi (project (path _map hi) (table-index rhs)))
             ;; build new map entry for newly discovered parse
             (nmap (inject 0 (path _map id) id))
             (nmap (inject nmap (path _map lo) lo))
             (nmap (inject nmap (path _map hi) hi)))
        ;; write new (parent) map entry
        (cwrite 1 maps nmap 0 0)))
    (on-success)
    (on-failure))
  ;; check to see if there's a map entry that exactly covers the input
  ;; and represents the start nonterminal (assumed to be id=0)
  (kernel check-result
    (wait)
    (on-entry)
    (read
      (bind input-count input-count exists preserve ())
      (bind maps maps exists preserve ((= (path _map id) 0)
                                       (= (path _map lo) 0)
                                       (= (path _map hi) (input-count i (path _tag))))))
    (body
      (cwrite 1 success 0 0 0))
    (on-success)
    (on-failure)))
(schedule
  (begin
    init-maps
    (while (and match (not check-result)))))))
```

Figure 53: Context free language recognizer (chart parser), matching

| cfl-recognizer | init | step | bkt-init | read | write | delete | succeed | fail | signal |
|---|---|---|---|---|---|---|---|---|---|
| program-root-123 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| init-maps | 1 | 24 | 46 | 46 | 23 | 0 | 1 | 0 | 1 |
| match | 10 | 32 | 601 | 601 | 22 | 0 | 10 | 0 | 10 |
| check-result | 10 | 11 | 11 | 11 | 1 | 0 | 1 | 9 | 10 |
| total | 21 | 68 | 658 | 658 | 46 | 0 | 12 | 11 | 22 |

Figure 54: CFL recognition statistics (23-element input)

pair of nonterminals.[32] We do not suggest that this is a particularly efficient way to recognize CFL strings, but it does demonstrate Greedy CAM techniques for composing adjacent parses.

Unlike deterministic FSM-recognition, where each input drives a single state transition, this problem allows both inputs (terminals) and nonterminals to participate in multiple parses. Just because a particular nonterminal instance $a$ is absorbed into a larger nonterminal instance $b$ does not mean that $a$ may not also be incorporated into a nonterminal instance $c$ at a later time. Thus, we are forced to maintain a representation of all reductions, as we are unable to establish that particular reduction has no future use.

The main problem here is that each pass will recompute all partial parse trees computed by the previous pass. Through the use of memoization (writing to "function"-mode tables) we can avoid storing the results twice (and then exploring them twice, yielding an exponential waste of effort), but we cannot avoid exploring them in the first place. This is a problem with bottom-up techniques that are not otherwise limited (by the structure of the grammar, as in LR(k) parsing, or by the incorporation of top-down techniques), not a problem with the Greedy CAM implementation of the algorithm.

The algorithm operates by iterating over a streaming pass that combines all adjacent parses for which a corresponding production exists. After each pass, we check to see if there is at least one map entry that covers the entire input, where that map entry is a valid instance of the "start" nonterminal. For the simple grammar representing a symmetric pattern of 0s and 1s, with a 2 in the middle, parsing a 23-element input requires 10 streaming passes (see Figure 54). The high pass-to-write ratio shows that very few new parses are found per iteration (and thus little useful streaming parallelism is achieved). This is largely due to the simple structure of the grammar; parse trees for this grammar will never have more than one nonterminal per tree layer, so the scope for parallelism is quite limited.

## 8.2 Tool commands and flags

To use LINT, start a Scheme system (Chez Scheme or Petit Chez Scheme) and load the system (`(load ''main.ss'')`. The loads the parser, interpreter, and examples from the TR.

### 8.2.1 Commands

- **to parse an example to intermediate form**:
  `(parse example-name)` or `(define intermed-name (parse example-name))`
- **to interpret intermediate form:**
  `(interp intermed-expr)`
- **to emit Dot representation of signal graph**
  `(graph-signals intermed-expr)`
- **to parse and interpret an example:**
  `(run example-name)`

### 8.2.2 Flags

The following flags can be set via `(set! flag-name flag-value)` or via fluid-binding.

- parser
  - `*parse-err-dump-size* ;; depth of expression stack displayed on parse error`
  - `*inline-empty-kernels* ;; enable/disable opt. of control flow`
- interpreter

[32] All context-free grammars can be transformed to this form, but exponential growth in the grammar size (and in the size of the resulting parse tree) may occur.

- `*min-interp* ;; suppress initial/final table dumps`
- `*noisy-interp* ;; dump lots of status info`
- `*debug-ti* ;; dump table-related activity`
- `*debug-osi* ;; dump query tuple (Operand Set) formation info`

## 8.3 Control structure implementation

This section describes the translation from LINT `schedule` expressions (as described in Figure 8) to `wait` expressions and `signal` actions.

As noted in the text, we distinguish between primitive and derived schedule expressions. The derived expressions are simply syntactic sugar for commonly-used primitive expressions, and are implemented via recursive tree transformation (macro expansion) to the appropriate primitives. For example,

```
(and <a0> <a1> ...  <an>)
```

where angle brackets and ... are meta-operators) translates to

```
(seq exit-fail <a0'> <a1'> ...  <an'>)
```

where the primed argument expressions represent the recursive transformation of the arguments. In this case, the short-circuiting `and` operation is implemented as a `seq` operator that executes its arguments in sequence, exiting after the first argument that returns failure (i.e., fails to process at least one query tuple). Similarly,

```
(do <a0> <a1> ...  <am> <an>)
```

translates to

```
(loop exit-post (not <an'>) (seq exit-last <a0'> ...  <am'>))
```

which executes all but the last argument in a do-while loop that exits when argument $a_n$ fails. We also perform some rewrites to reduce the number of cases handled by the processing of primitive schedule expressions. For example, we can avoid the need to process both the `and` and and `or` versions of the `par` operator by using DeMorgan's laws. The expression `(par and <a0> ...  <an>)` is rewritten to `(not (par or (not <a0'>) ...  (not <an'>)))`.

This leaves us with the task of processing expressions consisting of user kernel names, and `not`, `seq`, `par`, `loop`, `if`, or `raw` expressions. The processing is most easily explained as a hierarchical graph substitution algorithm, which we will specify using pictures.[33] In the graphical notation, a kernel is represented as a box with rounded corners. A connection at the top of the box represents the kernel's `wait` construct, while the two connections at the bottom of the box represent the signals raised by the `on-success` and `on-failure` constructs. All other schedule expressions are represented by a box with sharp corners. In this case, the single connection at the top of the box represents a single signal name which activates the box. The connections at the bottom represent signaling actions signifying the expression's success, failure, or termination (i.e., success or failure). A box may not have all three such connections, as some expressions (such as loops) do not admit notions of success or failure.[34]

Each scheduling operator is associated with one or more graphical templates, which specify how the operator's arguments (themselves recursively instantiated) and any introduced empty kernels are interconnected with each other and with the box's input and outputs. Empty kernels are used to perform all consolidation of signals, and are eliminated in a later phase.

[33] The actual implementation is somewhat more complex, but this explanation communicates the basic idea.

[34] We considered giving loops a semantics in which executing the body at least once means success, and failure to execute the body means failure. This is not very useful for "do" and "until" loops, which always execute the body. With "while" loops, we face an additional problem: since the predicate kernel is always the last to execute, we can't use its success or failure to generate the loop's result—it would always be failure. If, instead, we choose to use the body's success or failure to drive the loop result, we would have to store the value in memory, because the predicate (not the body) is always the last kernel to execute. Instead, we have chosen to have loops indicate only that they have completed, without any notion of success or failure.

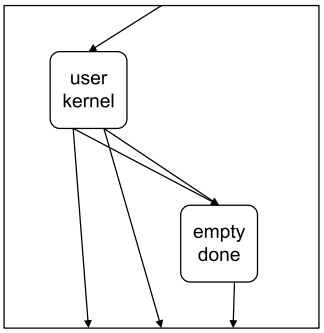


Figure 55: Representation of user kernel

- user kernels (Fig. 55): The box is invoked by raising the kernel's start signal. The box succeeds (resp. fails) if the kernel succeeds (fails), and completes on kernel success or failure. The template for `raw` schedule expressions is similar, but connects its first input (the "root" kernel) to the input, and its second input (the "leaf" kernel) to the outputs.
- `not` expression (Fig. 56): Negation is accomplished by switching the success and failure outputs.
- `if` expression (Fig. 57): The result of the predicate expression (which must have success and failure outputs) starts the consequent or alternative expression. Empty kernels are used to connect the consequent and alternative outputs to the `if` expression's outputs.
- `loop` expression (Fig. 58 shows the `exit-pre` case): An introduced empty loop header kernel fails into the predicate expression. If the predicate succeeds, the box's "completed" output is invoked; otherwise, the loop header is reinvoked. Note that the predicate expression must supply success and failure outputs.
- `seq` expression (Figs. 59 and 60 show the `exit-last` and `exit-succ` cases): In the `exit-last` case, completion of each subexpression drives the invocation of the next subexpression, with the final subexpression exiting the sequence box through the completion output. The `exit-succ` case is slightly more complex. The box's success output is driven by a concentrator kernel that "ors" the success outputs of the subexpressions, while the failure output derives directly from the failure of the final expression. The completion output is activated by any success (via the success concentrator) or by the final failure.
- `par` expression (Fig. 61 shows the `or` case): An introduced empty kernel starts both subexpressions. A failure concentrator "ands" the failure outputs, signifying failure and the completion of both subexpressions. Similarly, a completion concentrator waits for both subexpressions to terminate. The success concentrator's `wait` expression "ands" the termination signal with the "or" of the success signals; this implements a barrier that keeps control from exiting the box until all subexpressions have completed.

Rewriting terminates when all expression boxes have been replaced by kernel boxes. A separate pass "inlines" empty kernels where this is possible, given the CNF semantics of `wait` clauses. This separation eliminates a lot of special-case code in the rewriting process. In practice, empty kernels are unlikely to cause significant processing delays—the hardware scheduler can handle them as a special case (by directly implementing their failure actions, without performing kernel initialization). However, removing empty kernels does have significant cosmetic benefits, in that the resulting schedule graph is much smaller and easier to debug.

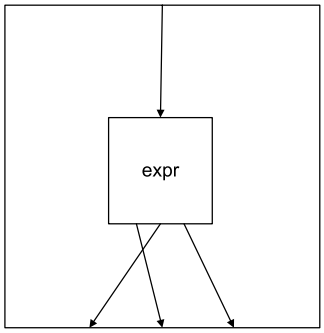


Figure 56: Representation of `(not expr)`

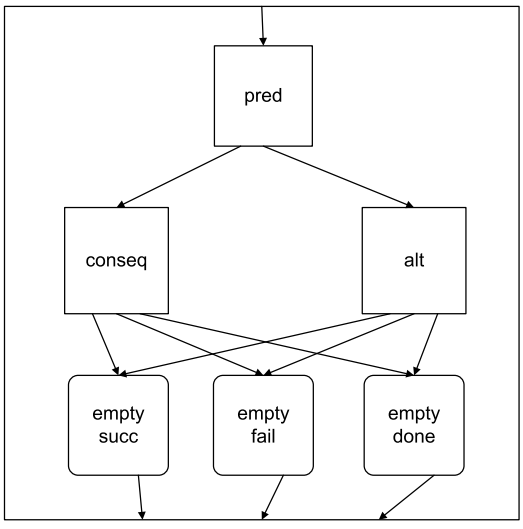


Figure 57: Representation of `(if pred conseq alt)`

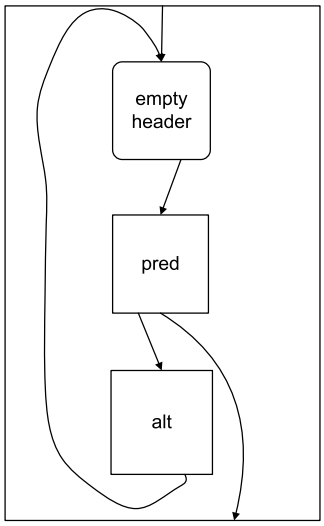


Figure 58: Representation of `(loop exit-pre pred body)`

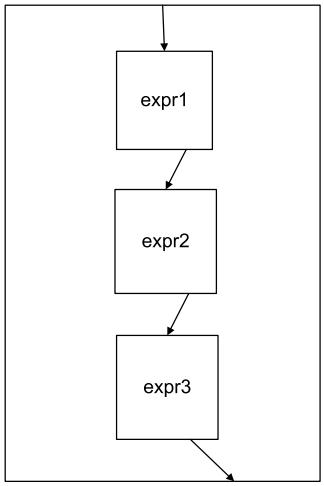


Figure 59: Representation of `(seq exit-last expr1 expr2 expr3)`

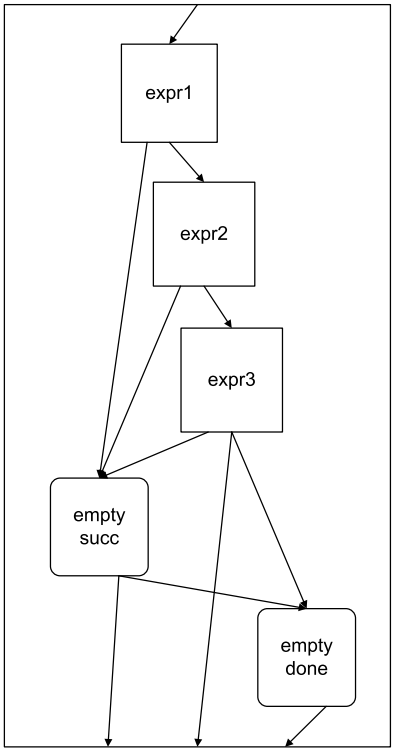


Figure 60: Representation of `(seq exit-succeed expr1 expr2 expr3)`

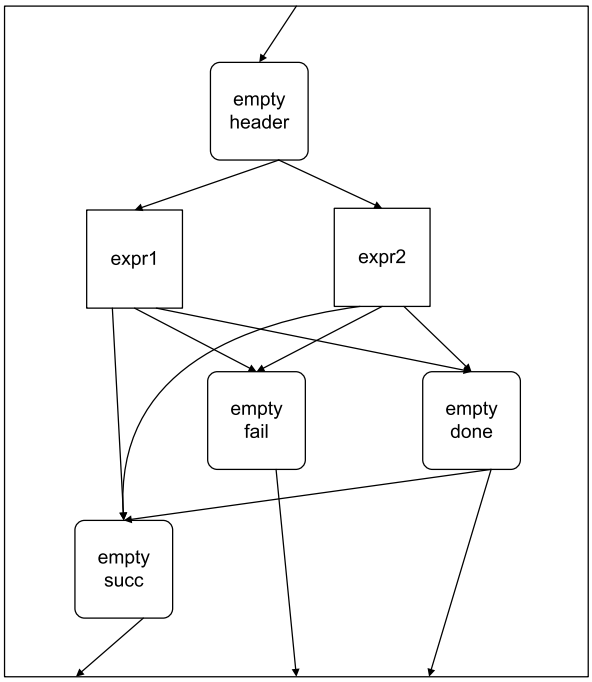


Figure 61: Representation of `(par or expr1 expr2)`

As a final example, we show the result of running the parser's graph dumper on the signal graph produced for the schedule

```
(schedule (begin
            input-to-maps-by4
            input-to-maps-by1
            (until
              (begin
                compose-by4
                (or
                  compose-by3
                  compose-by2
                  compose-by1)
                bump-index)
              exit-loop-on-done)
            eval-success))))
```

from `fsm-tree-by4`.

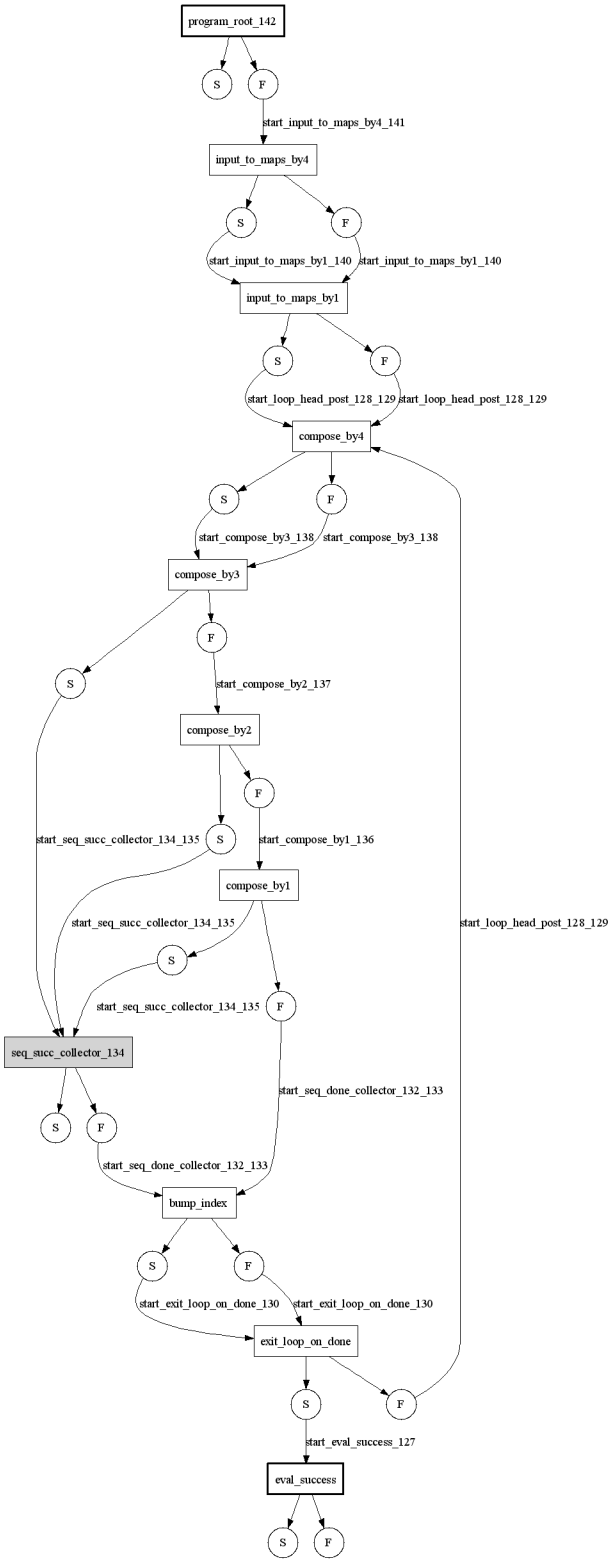


Figure 62: Schedule graph for `fsm-tree-by4`