Sadram

Sadram Arithmetic in C++

Robert Trout
Sadram, Inc.
Lititz PA USA
robertTrout@sadram.net

David Lynch
Sadram, Inc.
Lititz PA USA
davidLynch@sadram.net

ABSTRACT

The Sadram architecture (Symbolically Addressable DRAM) enables DRAM to sort incoming data as well as perform basic database access functions [6]. This is equivalent to processing ***symbolic addresses*** rather than the traditional ***linear addresses***. This extension enables DRAM to relieve the CPU of multiple DRAM accesses to perform elemental address computations and thereby save the time and energy that would otherwise be expended moving data to and from the CPU.

The consequences of moving computations into DRAM (process-in-memory or PIM) are profound. In this paper we describe the extensions that we have made to a standard C++ compiler to exploit such capability. We add a new organizational type, called a **sart**, to the language. A sart is ***<indexed>*** with angle brackets enclosing one or more expressions. The critical difference between arrays and sarts is that the sart index is not limited to small integers but may be a richer data element.

Sadram CONCEPTS

- Primacy of sort as an organizational tool
- Inherent parallelism of DRAM
- Address computation in DRAM
- Accessing arrays with non-linear indexes

KEYWORDS

- Symbolic memory addressing
- Sart (sorted array)
- DRAM addressing

1. Motivation

DRAM is intrinsically a block device, ie., accessing a single bit requires accessing an entire block (aka row). Sadram exploits this intrinsic parallelism.

About half of the energy consumed by memory is spent moving data to the CPU. This along with the intrinsic latency of DRAM, have driven CPU manufacturers to incorporate enormous cache memories into the CPU. Cache memory is fast and close to the CPU but has a much higher power/bit budget than DRAM because it is implemented as SRAM which draws power to merely hold the data. This is another motivation to exploit the parallelism of DRAM.

The motivation for extending the C++ compiler is to provide direct access to symbolic addresses. C++ was chosen because it does not have a native symbolic address mechanism. Python is also under active consideration; it only requires the modification of the NumPy and SciPy libraries to implement Sadram functionality.

Sadram efficiently implements sparse arrays without explicit coding because only those elements an array which are used incur the cost of storage (and indexing). Arrays with a large percentage of unused locations (sparse arrays) are stored more efficiently than traditional address based schemes.

Sadram functionality is implemented with very simple logic. This simplicity is not merely a recognition that memory transistors are not ideal for computations (memory transistors are designed to hold data, versus CPU transistors designed to move data), but simplicity is the inevitable consequence of high level parallelism. Only the most simple functions will be common to all algorithms.

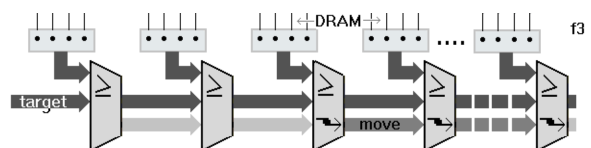
2. Sort and Symbolic Addresses

Sort is <u>*the most fundamental organizational tool*</u>. Access to unordered data requires O(N/2) operations to find one item in a collection of N item. Finding the same datum in an ordered set requires $O(\log_2 N)$ operations using simple binary chop methods and dramatically less with indexed methods. Given that sort is embodied in functions as diverse as database, code optimization, numeric accuracy control, and user display functions, it is not surprising that sort represents between 25% and 50% of *all* computations[3].

Sadram sorts data as it is written to DRAM without CPU involvement. Thus sort becomes 'free'. 'Inverting' the sort process, namely accessing data by its sorted key, is symbolic addressing.

3. Sadram Basics

Sadram adheres to the constraints of simplicity by restricting the functionality embedded in the DRAM to the compare operation and limiting data movement to short distances (ie., between neighboring cells). With these simple tools Sadram maintains the row buffer in sorted order. More elaborate structures maintain an entire block in sorted order[6].



4. Declaration of a Scalar Sart

Sadram introduces a new organizational type called a **sart** (acronym for a sorted array). The declaration of a sart resembles the declaration of a single dimensional array. A sart follows the normal scope rules of C++. The elements of the array can be a scalar (int, float, double, etc.) or a user defined structure (but not a class). The expression between [] defines the maximum number of elements in the sart. For scalars the word ‘sart’ following the type (int, float, etc.) signals that the array is sorted. For example (F4):
Elements of iSart can be referred to by position (iSart[0], iSart[1],

```
int sart iSart[1000]; F4
```

etc.) or by index (iSart<1234>, iSart<5678>, etc.). The positional mapping (ie., []) is *fluid* and can change when elements are added to or deleted from the sart. Positional mapping is used mostly for maintenance.

5. Sart Structures

The data element of a sart may be a structure (aka record). The word sart is followed by one or more *keys* within <>. Keys identify the fields within the structure which are symbolic indexes, eg., (F5):

```
typedef struct                                   F5
    {int  data;
     char akey[8];
    } STRUKT;
//declare sSart index by akey
STRUKT sSart  sart <akey> [1000];
```

Just like scalar sarts, each element of the sSart can be referred to by position or index, ie., sSart[0].data, sSart[1].akey, etc., or sSart<”elephant”>.data, etc.). The same ‘fluidity’ attaches to sart structures as to sart scalars (§4) when accessing a sart by position and index contemporaneously.

6. Accessing Fields within a Sart

Individual fields of a sart may be referred to by [position] or by <index>. This is the most elementary sart function; eg., (F6):

```
1. sSart<”football”>.data = 12;                  F6
2. sSart[0].data = 9;
3. int x = sSart<”elephant”>.data;
4. char mykey[8];
5. myKey = sSart<”nonsense”>.akey;
```

In line 1, the assignment to a sart field will either update the field if the record already exists or add a new element to the sart with the said key and data value if it does not exist. Changing an index will change the sorted order of the element and may change the apparent order of access by [position].

The important points are that these sart functions are implemented *by the DRAM* [6] and do not require multiple accesses by the CPU. Because they are implemented within the row buffer they have minimal effect on the normal DRAM read/write cycle.

7. Adding and Retrieving Entire Records

Entire records can also be added to a sart as follows (F7A):

```
sSart += {99,”football”};    F7A
sSart += {15,”elephant”};
sSart += {75,”nonsense”};
sSart += {13,”fatsnake”};
sSart += {19,”nonsense”};
```

As the records are added they are sorted by akey; the above five additions added to an empty sart results in:

```
sSart[0] == {15, ”elephant”}; F7B
sSart[1] == {13, ”fatsnake”};
sSart[2] == {99, ”football”];
sSart[3] == {75, ”nonsense”};
sSart[4] == {19, ”nonsense”};
```

8. Updating Sart Records

Updating a record can be done field by field (lines 1 and 2 in F8) or the entire record can be updated (line 3 in F8), as follows:

```
1. sSart[0].akey = ”elephant”;    F8
2. sSart<”football”>.data = 97;
3. sSart[1] = {13,”fatsnake”};
```

The record append operation (+= record) adds a new record to the sart, possibly creating a duplicate key in the process (§7A). The record replace operation (= record) overwrites the record if the specified record already exist. Both variants may change the key and the sequence of the sart.

9. Accessing Fields of a Sart Record

Records created by any one of the above operations can be accessed using the positional operator [position] or symbolically (F9):

```
1.sSart<”nonsense”>   .data = 75; F9
2.sSart<”nonsense”+0>.data = 75;
3.sSart<”nonsense”+1>.data = 33;
4.sSart<”nonsensX”-1>.data = 33;
5.sSart<-1>           .data = 75;
```

1. record.field at “nonsense” is set to 75. If the record does not exist, the operation will fault.
2. (+0) record.field at or immediately following the record at “nonsense” is updated.
3. (+1) record.field immediately following “nonsense” is updated.
4. (-1) record.field immediately preceding “nonsensX” (ie., the record <“nonsense”>) is updated; “nonsensX” merely sets the position; -1 aligns to the actual record.
5. <-1> record.field immediately preceding the ‘current record’ is updated.

10. The Mechanics of Field Access

Locating the record in the above operations requires accessing the 'more elaborate structures' mentioned in §3. After navigating these structures, the record itself is accessed and the appropriate field updated. The rows constituting the three levels of 'elaborate structures' are high priority residents of local DRAM cache and may not always be required to locate the record. [6]

A sart is always dynamically allocated even if the declaration is static or in the outer block of the program. The actual space occupied by the sart is less than the declared maximum when the sart is not fully populated. The actual space occupied by the sart will usually be larger than an equivalent array because of row allocation or indexing overhead but will often be smaller because the density of keys is low.

11. Sart Operations: ^sart

The indexes of a sart, written as ^sart, constitute a read only sart constructed from the keys of the sart operand. For example: after the five records have been added to the sart as shown in the §8, the indexes of the sart are (F11),

```
^sSart ==                                            F11
      {"elephant", "fatsnake",
       "football", "nonsense",
       "nonsense"}
```

No DRAM operations are involved in the ^sart operator; the sart so obtained is merely the index table from the sart itself.

12. Sart Operations: Sart versus Scalar

Normal scalar operators (designated as ⊕), ie., sart.field ⊕ scalar perform the operation on the fields within each record of each sart. The role of DRAM in this case is to generate and store the raw data field so that the CPU can perform the actual ⊕operation. Sart versus scalar operations are essentially the same as array operations of the same kind, ie., (F12),

F12

sSart.field ⊕ scalar

means **for all keys in ^sSart do**

sSart<key>.field ⊕= scalar

13. Sart Operations: Reduction of Sart to a Scalar

A scalar operator (eg., ⊕) can be applied to a sart field repeatedly to compute the 'sum' of the sart fields, ie., (F13),

F13

⊕ / sSart.field

means **for all keys in ^sSart do**

total = total⊕ sSart<key>.field

For example, if ⊕ == +, then + / sSart.data is the sum of the data of each element for all elements in the sart. See §17 for discussion of the order of the sart.

14. Sart Operations: Sart versus Sart

Scalar operators in which the left and right operands are sarts will apply the operator to each field of the sart, ie. sartL<key>.field ⊕ sartR<key>.field; for which both sarts have corresponding elements. Fields which are present on one sart but not the other are ignored, ie., F14),

F14

sartL.field ⊕ sartR.field

means **for all key in (^sartL ∩ ^sartR) do**

sartL<key>.field ⊕ sartR<key>.field

15. Compound Keys

A structure with two integer fields may construct its key by 'compounding' these two fields. The distinctive **!** conjunctive designates a compound key (F15),

```
typedef struct                                  F15
    {int    data;
     uint16 k1, k2;
     char   bfield[3];
    } COMP;
//declare sSart with compound key
COMP sSart <k1 ! k2> [1000];
```

The compound key is twice the size of the individual keys and constructed by packing the two keys in the upper and lower portions of a 32-bit word, ie., (k1 << 16) + k2. The sart will be indexed by two keys: k1 and k2.

16. Sart Operations: Dot Product

The two 'complementary' indexes are the foundation of dot product operations. Operators and sarts can be combined in a protocol that resembles the matrix multiply operation. The dot product operator is a combination of two scalar operators (say F and G) written F.G. When the dot product operator is applied to two sarts (eg., the operands sartL and sartR) it yields a third sartX with a combination of the fields from each sart (F16):

F16

sartX = sartL.field F.G sartR.field

means

sartX<i,j> = for all key in (^sartL ∩ ^sartR) do

F / sartL<key>.field G sartR<key>.field

The dot product protocol takes advantage of the fact that only elements that are assigned values in the parameters are incorporated in the computation. For sparse arrays, this can be an enormous saving. Ordinary matrix multiply is a special case of the dot product operator. Assuming each sart has two integer keys (say i and j), the second operator is applied to row (key i) of the left operand and column (key j) of the right operand, ie. sartL<i,k> is combined with all values in sartR for which sartR<k,j> is defined. The + operator is then applied to the vector of results.

17. Order of Sart Operations

The summation of ‘random’ floating point values is known to produce results which can have significant errors. These errors arise when a large number is added to a small number; the precision of the small number is washed out by the large number because the mantissa of the result has only a finite number of bits. Summation should be done from smallest absolute value to largest. This is achieved with negligible overhead by storing the intermediate results as a sart; the sart keeps the values in sorted order so the final summation can adhere to the proper summation method.

18. Meta Properties of a Sart

A variety of properties are associated with a sart and are available in a uniform way using the syntax **sart.propertyName**

Property Name	Meaning
RcdCount	Number of active records in the sart
MaxCount	Declared maximum count of sart
KeyName	Name of indexing field

19. Current State of Sadram Implementation

Sadram is currently implemented as a pre-processor to C++. We are currently investigating incorporating Sadram primitives into the numpy modules of Python.

20. Conclusion

Sadram introduces to C^{++} an organizational type called the sart which provides sorting and access to DRAM using symbolic addresses. These capabilities are built into the DRAM, which reduces CPU ⇔ DRAM traffic and thereby saves power & time in the execution of addressing functions. Access by symbolic address radically extends the capability of DRAM.

REFERENCES

[1] Doron Swade, 2001. *The Cogwheel Brain*, ISBN 978-0-349-11239-8, Abacus, London.

[2] R.L. Sites, 1996. “It’s the Memory Stupid!” *Microprocessor Rep*. 10, 10 (Aug. 1996).

[3] Donald E. Knuth, 1998. *The Art of Computer Programing*, *Volume 3*, 3, ISBN 0-201201-89685-0, Addison-Wesley, Boston, MA.

[4] Samuel K. Moore, 2021. “Memory Chips that Compute will Accelerate AI”, *https://spectrum.ieee.org/processing-in-dram-accelerates-ai*

[5] Onur Mutlu, Saugata Ghose, Juan Gomez-Luna, and Rachata Ausavarungnirun, “A Modern Primer on Processing in Memory”, *https://people.inf.ethz.ch/omutlu/pub/ModernPrimerOnPIM_springer-emerging-computing-bookchapter21.pdf*

[6] R. Trout and D Lynch-II, 2023. “Sadram: A new memory Addressing Protocol” *https://doi.org/10.1145/3631882.3631897*