Vector Processing on Scalar Architectures

Micah Beck and Antonio Castellanos
Department of Computer Science
University of Tennessee, Knoxville TN 37996-1301
{beck, castella}@cs.utk.edu

September 1994

Abstract

A 64-bit processor must necessarily implement substantial parallelism in the movement and processing of data. Data movement is inherently parallel at the bit level, and many operations implemented in the integer unit exhibit bit-level parallelism. A *microvector* is an array of small data items or *bit fields* packed into a single word. Scalar operations performed on a microvector can be used to implement vector parallelism on its constituent fields. In this paper we present **libuvec**, a library for computing with data which is packed into bit fields. We present experimental results showing excellent performance, up to 30 times the speed of scalar arithmetic in certain applications.

1 Introduction

The current generation of microprocessor architectures all have a word length of 64 bits, meaning that both instruction and data are processed 64 bits at a time: DEC Alpha, IBM/Motorola/Apple Power PC, and Intel Pentium. The increase in word length has accompanied an increase in the complexity of processor which can be implemented on a single chip. Longer instruction words can be used to allow more directly addressable registers or to specify operations for replicated functional units. Longer data words have thus far been used mainly to increase the range and precision of representable integers and floating point numbers.

A 64-bit processor must necessarily implement substantial parallelism in the movement and processing of data. Memory buses and internal data paths must all be 64 bits wide. Data movement is inherently parallel at the bit level, and many operations implemented in the integer unit exhibit bit-level parallelism. In this paper we will show how this data parallelism can be exploited using *microvector programming.*

A *microvector* is an array of small data items or *bit fields* packed into a single word. By storing bit fields in this way, the programmer can make full use of the available memory-to-register bandwidth and memory traffic can be reduced. In some cases, the total size of the data space is also reduced, improving the performance of the memory hierarchy.

There are two problems with this approach:

- packed bit fields may not correspond to addressable units of memory, and
- packed bit fields must be unpacked, processed, and repacked before being stored.

The first problem can be overcome by packing only addressable units: 8, 16, or 32 bits. This approach will work, but it limits the performance improvements which might be obtained by minimizing the size of bit fields. Packing in non-addressable units is most effective when applied to applications in which arrays are processed as aggregates and random addressing is not necessary.

The second problem is a serious one: in some applications the overhead of unpacking and repacking can be higher than the benefit of reduced memory traffic. In addition, the need to retain packed words while processing them increases register pressure.

Microvector programming is a technique for computing on packed bit fields *without unpacking them*. The key to our approach is that we exploit integer operations which are parallel at the bit level. We have obtained excellent results on several example applications, achieving a speedup of over 30 when implementing the Life cellular automaton.

Consider an array `A[1024]` containing integers which all lie in the interval $[0, 15]$. Each array element can be stored in a bit field of width 4, and so 16 of them can be packed into a 64-bit microvector. If a bit parallel operation such as ones complement is to be performed on the array, it can be performed just as well on the microvector with no unpacking. In this case 16-way parallelism reduces memory traffic and integer operations by a factor of 16, achieving linear speedup.

This observation can obviously be generalized to binary logical operations, and to addition on positive numbers. **Libuvec** is an extensive library of *vector parallel* operations, which includes logical, arithmetic, and some multiplicative operations on signed integers and fixed point numbers. In addition, **libuvec** provides support for data distribution of packed arrays which can increase the available parallelism. The **libuvec** source code and manual are available by anonymous FTP from directory `cs.utk.edu:pub/beck/libuvec`.

The remainder of this paper is organized as follows:

- in Section 2 we introduce the microvector data structure and vector parallel programming using **libuvec**,
- in Section 3 we explain the importance of data distribution when computing with microvector arrays,
- in Section 4 we illustrate microvector programming using two examples, and
- in Section 5 we present our conclusions, plans for future work on microvectors, and related work.

2 Microvector Programming

A microvector is a data word which is interpreted as a vector of bit fields. Any full word arithmetic operation has an interpretation as an operation on the corresponding microvector. The most useful operations are those which apply the same operation to every bit field. **Libuvec** is a collection of these *vector parallel* operations, implemented in C.

Definition A *microvector v of length l and width w* (or an *$l \times w$ microvector*) is a bit string of length lw, as illustrated if Figure 1. The *i'th element* of a $l \times w$ microvector v is the substring consisting of bits iw through $(i + 1)w - 1$.

Vector parallel operations which can be implemented by a single instruction yield linear speedup when applied to microvectors. The class of such operations is limited to data transfer, bitwise logical operations, and multiplication of an unsigned vector by an unsigned scalar. Unsigned addition and subtraction can be implemented by a single machine instruction in those cases where overflow is guaranteed not to occur.

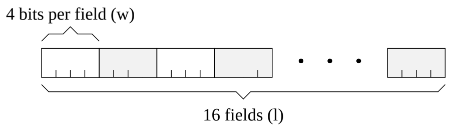


Figure 1: A 64-Bit 16×4 Microvector

A more general class of vector parallel operations can be implemented using a small number of machine operations. This class of operations includes signed and fixed point arithmetic, relational operators, and conditional expressions. Other operations such as testing for equality are more expensive, requiring a number of machine instructions proportional to the field width. General microvector programming is analogous to the programming techniques which have long been applied to pipelined vector processors [Hwa93].

Libuvec is a programming interface which implements a useful set of microvector operations. Some operations such as componentwise multiplication and division are best implemented by unpacking the argument vectors and repacking the result. These have been included in **libuvec** for completeness and programming convenience. The **libuvec** operations used throughout this paper section are representative of the complete interface, which is described in more detail in the **libuvec** manual [BC94].

2.1 The `uvec` type

Microvectors are represented using the type `uvec`, which is defined in header file `uvector.h` along with the rest of the **libuvec** interface. Normally, `uvec` will be defined as `unsigned long`. In the examples in this section, `u`, `v` and `x` are 16×4 microvector variables declared

```
uvec u, v, x;
```

Note that all microvector values have the same length in bits (`UV_SIZE`) and that the `uvec` data type applies to all microvectors regardless of their dimensions. The length of a microvector is therefore a function of the field width (`UV_LENGTH(`w`)`).

2.2 Simple Vector Operations

The simplest and most efficient microvector operations operate on integer values without overflow. The **libuvec** names given to these simple vector operations all start with the prefix `UV_` and take the field width as an argument.

Note that simple 4-bit fields can represent integer values in the range $[0, 15]$.

Example Bitwise logical disjunction $x_i = u_i \lor v_i$ is written

```
x = UV_OR(u, v, 4)
```

Example Equality $x_i = \begin{cases} 1 & \text{if } u = v \\ 0 & \text{otherwise} \end{cases}$ is written

```
x = UV_EQUAL(u, v, 4)
```

Example Average $x_i = (u_i + v_i)/2$ is written

```
x = UV_RSHIFT(UV_PLUS(u, v, 4), 4)
```

as long as the addition is guaranteed not to overflow the 4-bit field.

Example The conditional expression $x_i = \begin{cases} u_i & \text{if } u_i = v_i \\ 0 & \text{otherwise} \end{cases}$ is written

```
x = UV_IF(UV_EQUAL(u, v, 4), u, UV_S2VEC(0, 4), 4)
```

where `UV_S2VEC()` converts a scalar field value s to a vector in which every field holds the value s.

2.3 Overflow Vector Operations

In order to avoid arithmetic overflow, k-bit signed integer operations can be implemented using fields of width $k+1$ and requiring that the most significant always be 0. Field overflow due to addition will cause the most significant bit to be set to one, but it can be reset between operations. The cost of this encoding is one bit per field plus one masking operation for every operation which can cause overflow.

Libuvec includes a set of operations which implements this "overflow" representation of signed microvector fields. The **libuvec** names given to these simple vector operations all start with the prefix `UVO_` and have a final argument which specifies the total width of the field, including the overflow bit.

Note that 4-bit overflow fields can represent integer values in the range $[-4, 3]$.

Example Subtraction $x_i = u_i - v_i$ is written

```
x = UVO_MINUS(u, v, 4)
```

Example Average $x_i = (u_i + v_i)/2$ is written

```
x = UVO_RSHIFTA(UV_PLUS_(u, v, 4), 4)
```

Example The less-than relation $x_i = \begin{cases} 1 & \text{if } u_i < v_i \\ 0 & \text{otherwise} \end{cases}$ is written

```
x = UVO_LT(u, v, 4)
```

Example The sign function $x_i = \begin{cases} 1 & \text{if } u_i < 0 \\ 0 & \text{otherwise} \end{cases}$ is written

```
x = UVO_SIGN(u, 4)
```

2.4 Fixed Point Operations

Because the floating point unit is not parallel at the bit level, microvectors with floating point fields cannot be efficiently implemented. In contrast, operations on microvectors with fixed point fields can be implemented using a small number of integer operations. **Libuvec** provides support for microvector programming with fixed point fields.

A fixed point number $m2^f$ has two parts, an integer mantissa m and an integer exponent f. The **libuvec** fixed point representation stores only the mantissa. Additive operations on fixed point number with the same exponent can be implemented using integer arithmetic, as long as all arguments have the same exponent. Multiplicative operations require that the exponent be supplied as an argument. **Libuvec** currently supports fixed point only operations whose arguments and result all have the same exponent, which must be negative.

In order to obtain the full accuracy available in the result, the result of multiplication must be computed using twice the field width of the operands. This requires the vector to be split into two subvectors consisting of the fields with even and odd indices and twice the field width. The results of multiplication are then truncated and recombined to form a single vector.

In this example, we will assume that `u,v` and `x` are microvectors with 4-bit fixed point fields and exponent $f = -2$. and `s` is a scalar integer. Note these fixed point fields can represent integer multiples of .25 in the range $[-1.00, .75]$.

Example Subtraction of fixed point numbers $x_i = u_i - v_i$ is written

```
x = UVO_MINUS(u, v, 4)
```

Example Multiplication of an integer scalar by a fixed point vector $x_i = su_i$ is written

```
x = UVO_XSMUL(UVO_I2X(s, -2), u, 4, -2)
```

2.5 Field Access Operations

In cases where computation cannot be efficiently implemented in vectorized form, and for the purpose of external reading and writing of data it is necessary to access individual field values. When accessing signed field values, truncation and sign extension must also be performed as appropriate. Field access operations are defined on vector values, and also on microvectors stored in memory.

The abstraction of arbitrary length arrays of field values implemented using microvectors is also supported. The distribution of data (block vs. scattered) in these arrays of microvector fields is very important in exposing vector parallelism in the presence of loop carried data dependences. This issue is discussed more fully in section 3.

Example The value $s = u_1$ is written

```
s = UVO_FGET(u, 4, 1)
```

Example The value of u with field u_1 set to -1 is written

```
s = UVO_FSET(u, 4, 1, UVO_VAL(-1, 4))
```

Example If `uvec *up` is a pointer to a microvector stored in memory, then value $s = *\mathtt{up}_1$ is written

```
s = UVP_GET(up, 4, 1)
```

Example If `uvec ua[]` is an array of microvector fields stored in scattered distribution, then field $s = \mathtt{ua}[253]$ is written

```
s = UVA_SGET(ua, 4, 253)
```

2.6 Inlining

The **libuvec** interface is defined in terms of macros, as indicated by the use of upper case names. The library is in fact implemented in two forms: subroutines and inlined macros. The names of the subroutine library are identical to the the macro names, but written in lower case.

The default mode of compilation uses subroutines rather than macros by defining the upper case names to be the lower case ones. This gives the greatest generality in programming the library, reduces the chance of an error due to inlining, and eases the interpretation of compiler errors and debugging. Unfortunately, introducing subroutine calls for basic microvector operations makes microvector code terribly slow.

In order to obtain high performance, **libuvec** calls must be inlined and the resulting code must be optimized. Inlining is accomplished by simply defining the symbol `UV_INLINE` before including the file `uvector.h` in your source file. The inlined code is full of constant expressions and should be optimized aggressively. Unfortunately, inlining does somewhat restrict the manner in which the library can be used.

The inlined library makes extensive use of token concatonation, a feature of the `gcc` preprocessor which other compilers do not implement. This has two implications: the inlined code must be complied using `gcc`, and field widths must be *literal* constants. This means that the following code sequence is *not* acceptable:

```
#define UV_INLINE
#include "uvector.h"
#define W    4

v = UV_NOT(u, W);
```

because the call *must* be written `UV_NOT(u, 4)`. Not every macro uses concatenation, and those that do not can be called with an arbitrary expression for the field width.

There are some ways around this problem:

- In order to implement a global symbol `W` to specify the field width for an entire program, use a preprocessor such as `sed` to textually substitute its literal value into the code before compilation.

- Individual subroutine calls can be explicitly inserted in inlined code by using the lower case subroutine names, but this may slow execution drastically.

Future versions of the library may remove the dependence on `gcc` and implement other solutions to this problem.

3 Data Distribution: An Architectural Analogy

In Section 2 we saw how an array of field values can be stored as a microvector array. When implementing microvector arrays, fields can be distributed across the array in one of two ways: a *scattered* or a *blocked* distribution. The terminology is taken from the field of distributed memory multiprocessing because one way of understanding microvector computing is to use an architectural analogy. We view a scalar computer operating on microvectors to be a SIMD (single instruction, multiple data) machine with distributed memory [Hwa93].

When a vector parallel operation is performed on a 16×4 microvector, then the same operation is performed on every field. Fixing the width of the field, the scalar computer can be viewed as a SIMD machine with 16 4-bit processors. The resources of each processor are a 4-bit wide slice of scalar processor's registers, data paths and processing logic. Vector parallel operations do not communicate between these "processors."

In this analogy the address space of the scalar processor is also divided into 16 slices, one slice being local to each "processor". A single address does not specify a single location, but specified one location in each memory slice. Memory operations act on local registers and local memories in parallel. SIMD computing and addressing of this sort is found, for example, on the vector units of the Thinking Machines Corp. CM-5 [Hwa93].

In the context of this analogy, scalar instructions can be interpreted in a multiprocessor context. Bit parallel operations implement local computation, shifts are for parallel communication between processors, and multiplication by a scalar uses a built-in broadcast mechanism. The analogy is most useful for understanding the relevance of data distribution to microvector computing.

3.1 Scattered Distribution

Scattered distribution is the most direct implementation of a microvector array. Adjacent array elements are stored in adjacent fields, starting with the first microvector in the array and proceeding to the last, as illustrated in Figure 2. Adjacent elements are stored in different local memories and are therefore local to different processors.

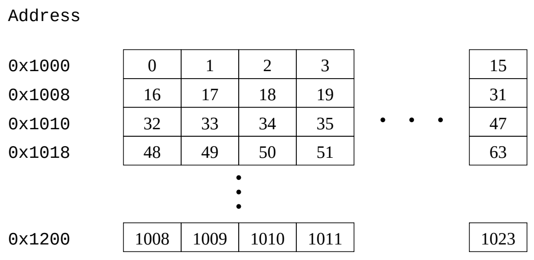


Figure 2: 1024-Element Array: 64 16×4 Microvectors, Scattered Distrib.

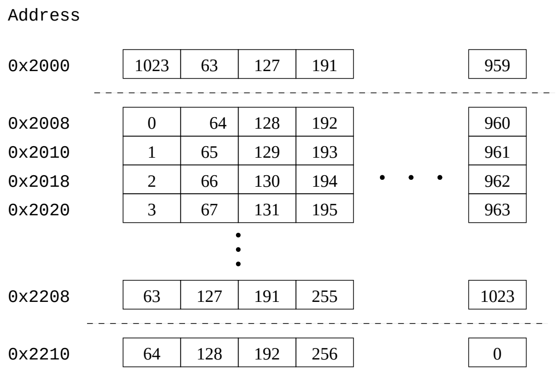


Figure 3: 1024-Element Array: 64 16×4 Microvectors, Blocked Distrib., Replicated Boundary

Example Consider computing the sum of two arrays A and B implemented as arrays of microvectors with a scattered data distribution. To compute the sum of these two arrays it is simply necessary to add the corresponding vectors:

```
uvec A[64];

for (i = 0; i < 64; i++)
  C[i] = A[i] + B[i];
```

A scattered data distribution can be conveniently used whenever the indices of arguments to binary operations are the same, or differ by a multiple of the vector length. Specifically, binary operations cannot be conveniently performed on adjacent array elements. This restriction rules out many applications.

3.2 Blocked Distribution

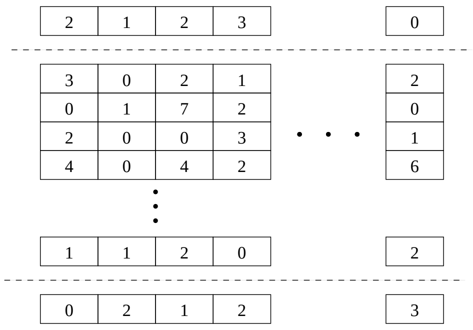
In order to allow binary operations to be performed on adjacent array elements, it is necessary to store them at adjacent addresses in the *same* local memory. This means that array elements are stored in the first field of every microvector in the array, then in the second field, etc, as illustrated in Figure 3. Adjacent microvectors in the array then hold vectors of adjacent elements, except at the beginning and end of each local memory where a special case occurs.

In many cases, the boundary condition can be handled by maintaining a copy of the values at the extreme end of each memory in a location at the opposite extreme end of the next memory. **Libuvec** implements microvector arrays with a replicated boundary and provides a boundary update function.

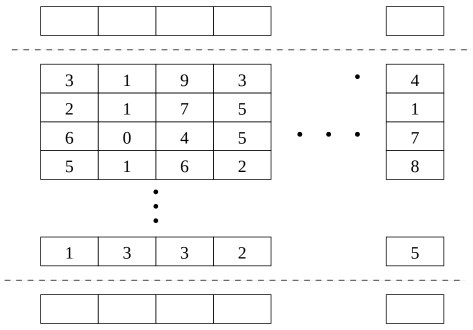
Example Consider computing the sum $B[i] = A[i] + A[i+1]$ of adjacent elements where arrays A and B are implemented as an array of microvectors with local data distribution and with a replicated one-field boundary. Let us assume that a copy of the boundary of A is maintained. Then to compute the sum of $A[i]$ and $A[i+1]$ it is simply necessary to add the corresponding vectors. The replicated boundary of B must then be updated.

```
uvec A[66], B[66];

for (i = 1; i < 65; i++)
  B[i] = A[i] + A[i+1];

UV_BWRAP(B, 64, 4, 1);
```

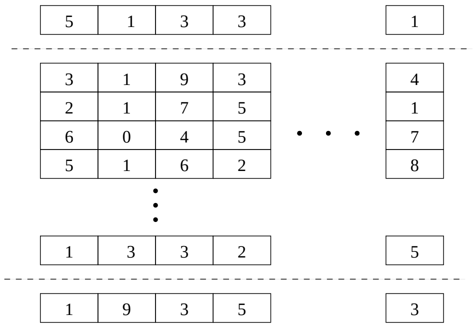
The following data illustrate the execution of this code. The values shown in the array cells below are arbitrarily chosen data items, not cell indices as in Figures 2 and 3. Note that the values in the boundary of B are not defined until after the call to UV_BWRAP().



1. Array A before executing loop



2. Array B before executing UV_BWRAP()



3. Array B after executing UV_BWRAP()

When the entire array is not updated at once, replicating the entire boundary after each update may not be feasible. In this case, more complex code which performs communication between memories can be used.

4 Programming Examples and Performance

4.1 Matrix Multiplication

Consider a one-dimensional array `A[1024]` of 8-bit unsigned integer values. This array can be treated as an array of byte values or as an array of 128 8×8 microvectors. If overflow is guaranteed not to occur, then a microvector `u` can be multiplied by a positive scalar value `s` by writing `UV_SMUL(s, u, 8)`. The entire array `A` can be multiplied by s in a simple loop using byte arithmetic:

```
unsigned char A[1024], B[1024];

for (i = 0; i < 1024; i++)
  B[i] = s * A[i];
```

If `A` is treated as an array of microvectors, this loop can be strip mined and microvectorized:

```
uvec A[128], B[128];

for (i = 0; i < 128; i++)
  B[i] = UV_SMUL(s, A[i], 8);
```

Because the `UV_SMUL()` operation is implemented by a single integer multiplication, we obtain a speedup of 8 over byte arithmetic using vectors of length 8, or *linear speedup.*

Unoptimized matrix multiplication can be implemented in byte arithmetic as a simple nesting of loops:

```
unsigned char A[1024][1024], B[1024][1024];
int i, j, k;

for (i = 0; i < m; i++)
  for (j = 0; j < m; j++)
    for (k = 0; k < m; k++)
        B[i][j] += A[i][k] * A[k][j];
```

A two-dimensional byte array can be viewed as a two dimensional array of microvectors by vectorizing in one dimension. We can write a vectorized form of matrix multiplication using scalar-vector multiplication and vector addition as basic operations:

```
uvec A[1024][128], B[1024][128];
int i, j, k1, k2;

for (i = 0; i < 1024; i++)
  for (k1 = 0; k1 < 128; k1++)
    for (k2 = 0; k2 < 8; k2++)
    {
      int s = UV_FGET(A[i][k1], w, k2);

      for (j = 0; j < 128; j++)
        B[i][j] = UV_PLUS(C[i][j], UV_SMUL(s, A[k1*8+k2][j], 8), 8);
    }
```

Performance

Figures 4 and 5 show the execution time and speedup respectively for multiplying two 1024×1024 matrices in various ways. Our experiments involved several similar versions of matrix multiplication:

- using scalar byte arithmetic (`mmulb`),
- using floating point arithmetic (`mmulf`),
- using 8×8 integer microvectors to store the matrix and unpacking them to use scalar arithmetic when performing arithmetic operations (`mmulp`),
- using 8×8 integer microvectors (`mmulv`),
- using 8×8 fixed point microvectors (`mmulx`).

The total number of bits in each microvector was varied to obtain a speedup curve: n = 8, 16, 32, and 64 bits. Speedup is defined to be the ratio of execution times to the execution time of `mmulf`. The independent axis of these graphs shows the number of bits in a microvector. The line labeled "linear speedup" shows where the speedup is equal to the length of the microvector.

The speedups achieved with 64-bit microvectors are given in Figure 6.

- `mmulp` packs byte values into microvectors but unpacks in order to perform byte arithmetic. Its performance is worse than any scalar version.
- In our experiments, the speed of `mmulf` is 1.7 times that of `mmulb`, which uses scalar integer arithmetic. So although the performance of `mmulv` represents 89% of linear speedup over `mmulb`, this is only 52% of linear speedup over `mmulf`.
- Finally, `mmulx` uses fixed point scalar-vector multiplication, which is the most expensive **libuvec** arithmetic operation, and so achieves only 27% of linear speedup of `mmulf`.

In all cases, the code has been optimized by hand to obtain the best possible performance. All experiments were performed on the DEC Alpha 3000/500 with 222 Mbytes of main memory.

It is interesting to note that the execution time for `mmulf` is approximately half that of an equivalent program which uses integer arithmetic. This difference reflects the relative speed of integer and floating arithmetic.

4.2 Cellular Automata

A 2-dimensional cellular automaton (CA) is a discrete simulation of a world in which locations are arranged in a two-dimensional grid of cells [vN66, Ula86]. Every cell of the grid c is associated with an integer value. The execution of a CA proceeds in synchronous steps known as *generations*. In each generation, the value of every cell in the grid is updated. The new value $c1[i,j]$ of a cell $c[i,j]$ is a function of the old values of the grid in some fixed *neighborhood* of c.

Parallel execution of cellular automata has been implemented both in on general parallel processors and using specialized hardware [TM87, BH93]. The most well-known cellular automaton is Conway's 2-dimensional CA "Life" [Gar70, Gar71]. Life is binary: the value associated with every cell is either 0 or 1. The Life update function is defined by this rule:

$$c1[i,j] = \begin{cases} c[i,j] & \text{if } s = 2 \\ 1 & \text{if } s = 3 \\ 0 & \text{otherwise} \end{cases} \qquad \text{where} \qquad s = \sum_{\substack{x,\,y\,\in\,\{-1,0,1\} \\ x \neq 0 \;\vee\; y \neq 0}} c[i+x, j+y]$$

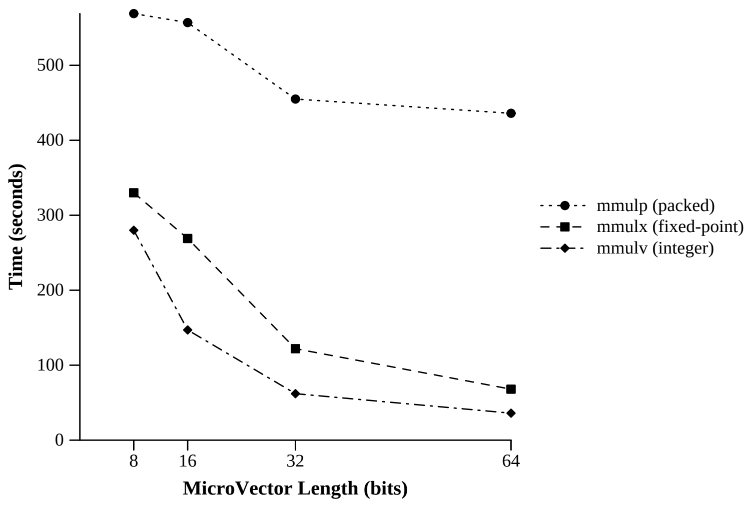


Figure 4: Total computation time: Multiply two 1024×1024 matrices

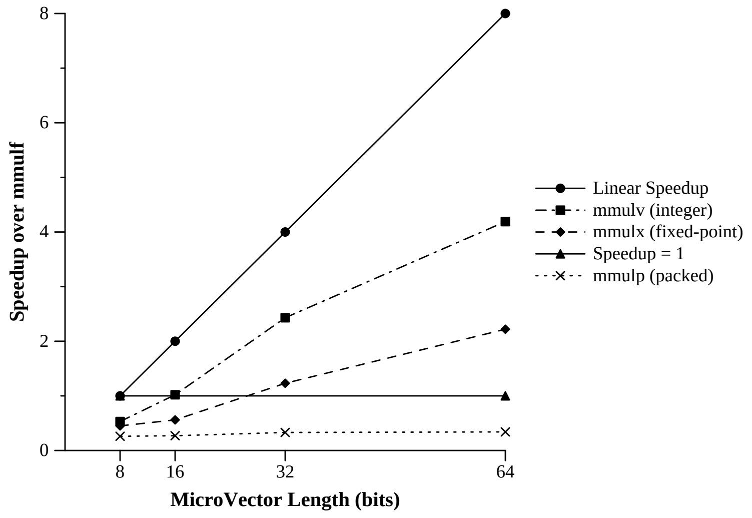


Figure 5: Speedup Over Scalar Arithmetic: Multiply two 1024×1024 matrices

Program	`mmulp`	`mmulv`	`mmulx`
1024×1024 matrices	0.34	4.19	2.22

Figure 6: Matrix Multiply Speedup Using 64-bit Microvectors

Implementing The Grid

The most straightforward implementation of a 2D grid is as a two-dimensional array of binary values. Consider a 4096×4096 grid, implemented using 64-bit microvectors. Since each cell of the grid holds either a zero or a one, it is natural to consider using 64×1 microvectors. However, in the calculation of the update function, it is necessary to sum the values of eight grid locations, and the resulting sum can require up to four bits to represent.

We will show how to implement the update function using 16×4 microvectors. The best performance is obtained by storing the grid using 64×1 microvectors, unpacking each of these into 4 16×4 microvectors in order to compute the update function, and then repacking these back into a single 64×1 microvector. Figures 7 and 8 gives execution time and speedup for both microvector programs.

Each row of the grid contains 4096 cells and so can be stored in 256 16×4 microvectors. Because the sum operation references adjacent locations in the grid, a blocked data distribution is used.

Implementing The Update Function

Having chosen a data structure for the 2D grid, it remains to implement the calculation of generations. If we let `c[4096][256]` represent the current state of the grid, then we calculate its state one generation later as the array `c1[4096][256]`. We update each element using an outer loop which iterates over rows and an inner loop which iterates over the microvectors in the elements of each row. Thus the columns in each row are updated 16 at a time.

```
for (i = 0; i < 4096; i++)
    for (j = 0; j < 256; j++)
    {
        s = UV_PLUS(c[i][j-1], c[i][j+1], 4);
        s = UV_PLUS(s, c[i-1][j-1], 4);
        s = UV_PLUS(s, c[i+1][j-1], 4);
        s = UV_PLUS(s, c[i-1][j], 4);
        s = UV_PLUS(s, c[i+1][j], 4);
        s = UV_PLUS(s, c[i-1][j+1], 4);
        s = UV_PLUS(s, c[i+1][j+1], 4);

        c1[i][j] = UV_IF(UV_EQUAL(s, S2VEC(2, 4), 4), c[i][j],
                     UV_IF(UV_EQUAL(s, S2VEC(3, 4), 4), S2VEC(1, 4),
                     0), 4);
    }
```

The update function has two parts, the calculation of the sum and the conditional. Since every cell location takes the value zero or one, there is no possibility of overflow when adding eight of them in a vector element four bits wide. Thus, sum is microvector parallel and can be implemented with `UV_PLUS()` and the case split is implemented using nested conditionals.

Performance

Figure 7 shows the time taken to calculate 100 microvectorized updates of a 4096×4096 grid:

- using byte arithmetic (`lifeb`),
- using 64×1 microvectors to store the grid and integer arithmetic to compute the update function (`lifep`),
- using 16×4 microvectors to both store the grid and to compute the update function(`lifev4`), and
- using 64×1 microvectors to store the grid and 16×4 microvectors to compute the update function (`lifev1`).

The time measured was only the time taken to actually compute the updates. It does not include the time taken to read the initial data and pack it into microvectors, and or the time taken to unpack the final data and write it out.

Figure 8 shows the speedup achieved by the microvector implementations over byte arithmetic. The independent axis shows the length of microvector used in bits. Speedup is calculated as the time taken for the microvectorized computations divided by the time taken by `lifeb`.

Two lines labelled "linear speedup" show where the speedup is equal to the length of the microvector.

- The speedup curve for `lifev4` can be compared directly with the 4-bit linear speedup curve, and it fits closely.
- In `lifev1` 1-bit fields are used for memory operations and 4-bit fields are used for computations. As would be expected, the observed performance in lies between the linear speedup curves for 1-bit and 4-bit fields,
- `lifep` shows relatively low speedup, showing that in this case the overhead of unpacking and repacking data is significant.

The speedups achieved with 64-bit microvectors are given in Figure 9, for two grid sizes.

- `lifep` packs byte values into microvectors but unpacks into order to perform byte arithmetic. It achieves moderate speedup over `lifeb`.
- The maximum speedup of 27 occurs for `lifev1` on the larger grid. This figure reflects both microvector parallelism and the effect on the memory hierarchy of an 8-fold reduction in data memory size.
- `lifev4` achieves higher speedup on the smaller grid than on the larger grid may also be due to a memory hierarchy effect.
- The speedup achieved on at 1024×1024 grid yields a speedup curve much closer to 4-bit linear.

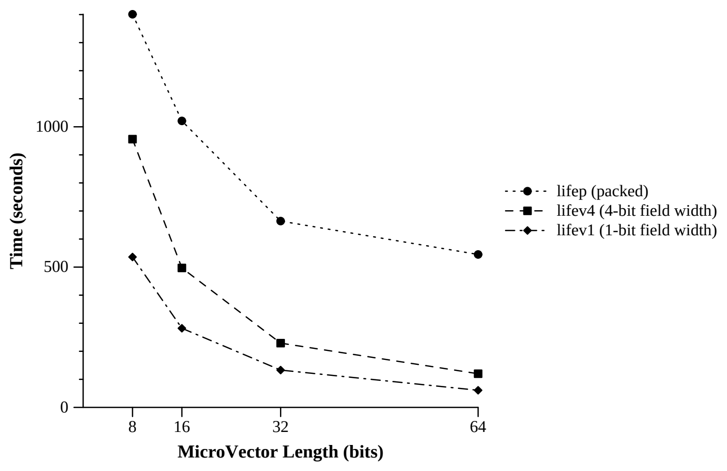


Figure 7: Total Computation Time: 100 Life updates, 4096 × 4096 grid

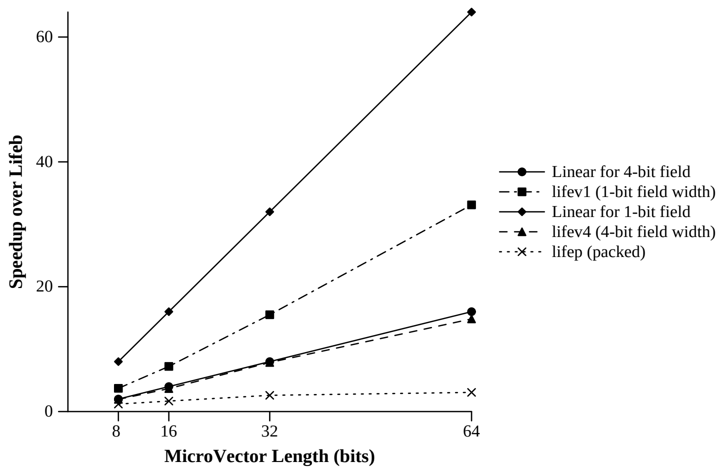


Figure 8: Speedup Over Byte Arithmetic: 100 Life updates, 4096 × 4096 grid

Program	`lifep`	`lifev4`	`lifev1`
1024 × 1024 grid	2.48	16.40	20.50
4096 × 4096 grid	3.07	14.84	33.11

Figure 9: Life Speedup Using 64-bit Microvectors

5 Conclusions, Future and Related Work

In this paper, we have introduced the notion of microvector programming and have described the **libuvec** programming interface. We have shown how two example applications can be easily coded using **libuvec**:

1. Dense matrix multiplication shows speedup over 4 when operating on 8-bit integers and over 2 when operating on 8-bit fixed point numbers. These results show that excellent microvector performance can be difficult to achieve when compared to floating point operations which are highly optimized in hardware. Our results indicate that some numerical applications can be speeded up using microvector programming.

2. The Life cellular automaton (CA) shows speedup over 30 when data are packed into microvectors as tightly as possible. Other applications for cellular automata include discrete physical and biological simulations.Our results indicate that some of these applications can be significantly speeded up using microvector programming.

Future Work Our continuing research on microvectors has two directions:

1. We are finding useful applications for the **libuvec** library. Microvector programming shares with all vector processing the weakness that it cannot be used in every application. Promising candidates for microvector programming include linear algebra, discrete simulation, text processing and graphics.

2. We are developing automatic tools for translating parallel programs into microvector form.
   (a) Vector C is a preprocessor for the C language which translates high level vector operations into microvector code using **libuvec**. A Vector C can always correctly interpreted as standard C, so Vector C code can be written, debugged, and executed in the standard C programming environment.
   (b) Cellang is a simple language for describing cellular automaton developed by Dana Eckart of Radford University [Eck92]. Cellang is inherently very parallel, and so provides a good platform for automatic vectorization. We are developing a back end for the Cellang compiler which generates Vector C code.

The **libuvec** source code and manual are available by anonymous FTP from directory `cs.utk.edu:pub/beck/libuvec`. We are very interested in exchanging ideas and collaborating with researchers in any field where microvector programming can be applied. **Libuvec** is an ongoing development project, and its future development will be directed by the needs of users.

Related Work Bitwise logical operations have long been used on bit arrays in many applications, most notably bit-mapped graphics processing. Microvector programming is a generalization of "multispin coding", a technique developed by Jacobs and Rebbi for implementing CA-based simulations [JR81]. The implementation of signed integer and fixed point operations are original to our work.

Acknowledgements Dana Eckart's development and distribution of Cellang led us to consider general techniques for parallelizing the execution of CA programs. Thanks to Bruce Boghosian for his early encouragement and for bringing multispin coding to our attention. We are grateful to Todd Letsche for his helpful comments on this paper.

References

[BC94] Micah Beck and Antonio Castellanos. *Libuvec Reference Manual.* University of Tennessee, September 1994. Distribution by FTP from `cs.utk.edu:pub/beck/uvector`.

[BH93] Per Brinch Hansen. Parallel cellular automata: A model program for computational science. *Concurrency: Practice and Experience*, 5(5):407–423, August 1993.

[Eck92] J. Dana Eckart. *Cellang 3.0: Language Reference Manual.* Radford University, April 1992. For distribution contact `dana@rucs.faculty.cs.runet.edu`.

[Gar70] Martin Gardner. The fantastic combinations of John Conway's new solitare game "life". *Scientific American*, 223(10):120–123, October 1970.

[Gar71] Martin Gardner. On cellular automata, self-reproduction, the garden of eden and the game "life". *Scientific American*, 224(2):112–117, February 1971.

[Hwa93] Kai Hwang. *Advanced Computer Architecture.* McGraw-Hill, New York, NY, 1993.

[JR81] Laurence Jacobs and Claudio Rebbi. Multi-spin coding: A very efficient technique for monte carlo simulations of spin systems. *Journal of Computational Physics*, 41:203–210, 1981.

[TM87] Tommaso Toffoli and Norman Margolus. *Cellular Automata Machines: A New Environment for Modelling.* The MIT Press, Cambridge, MA, 1987.

[Ula86] S. Ulam. *Science, Computers and People: From the Tree of Mathematics.* Brikhäuser, Boston, MA, 1986.

[vN66] John von Neumann. *Theory of Self-Reproducing Automata.* University of Illinois Press, Urbana, IL, 1966. Edited and completed by A. W. Burks.