APPLICATION-SPECIFIC MEMORY INTERLEAVING FOR FPGA-BASED GRID COMPUTATIONS: A GENERAL DESIGN TECHNIQUE*

Tom VanCourt and Martin Herbordt

Boston University, Department of Electrical and Computer Engineering
8 St. Mary's St., Boston MA USA 02215
email: {tvancour, herbordt} @ bu.edu

ABSTRACT

Many compute-intensive applications generate single result values by accessing clusters of nearby points in grids of one, two, or more dimensions. Often, the performance of FGPA implementations of such algorithms would benefit from concurrent, non-interfering access to all points in each cluster. When clusters contain dozens of points and access patterns are irregular, multiported memories are infeasible and vector-oriented approaches are inapplicable. Instead, the grid points may be distributed across multiple interleaved memory banks so that, when accessing any cluster, each point comes from a different memory bank. We present a general technique based on the application's multidimensional indexing rather than linearized memory addresses. This technique maps the cluster structure into a custom, interleaved memory using the FPGA's multiple on-chip RAMs and configurable data paths. Case studies demonstrate rectangular and non-rectangular grids of different dimensions, including performance vs. resource tradeoffs when cluster sizes are not powers of two.

1. INTRODUCTION

FGPA accelerators are entering the main stream of high performance computing. Researchers have demonstrated impressive performance gains using FPGA acceleration, 100-1000× in some applications. These applications have generally been hand coded by skilled logic designers, however. They are generally built as point solutions to particular problems or families of problems. Few if any techniques are available for the general problem of applying FPGAs to arbitrary computing problems in ways that use the unique features of FPGAs to their full potential. In contrast, the last fifty years of software development, with millions of developers worldwide, has a large body of accepted techniques, applicable to computing problems of many kinds.

This paper presents a configurable family of memory structures useful for many applications in FPGA-based computing. It is applicable to problems in solid modeling and visualization, computational chemistry, cellular automata, discrete solutions to differential equations, and many other staples of high-performance computing. Such applications may have idiosyncratic computation pipelines, but share one common feature: each output value depends on input from a cluster of neighboring grid points, where dimensionality of the grid and the neighborhood of points in a cluster vary by application. In the applications of interest, computation throughput would be improved if all points in the cluster could be fetched in one memory cycle. The contributions of this work create a step by step process for mapping application-specific reference patterns to memory structures with application-specific interleaving. The resulting memory stores grid points non-redundantly in multiple RAMs, allowing single-cycle, parallel access to all points in any one cluster. This kind of interleaving takes advantage of the FPGA's unique resources. It uses dozens or hundreds of the independently addressable RAMs in the FPGA, it can be configured according to the unique characteristics of each computation, and it relies on the FPGA's native ability to create custom interconnection networks with data path widths into the thousands of bits.

This technique for application-aware memory interleaving is described as follows. First, we present a number of applications built around computations of the kind being addressed, followed by a brief survey of related work in traditional memory structures and in FPGA-based computing. Next, we present case studies of interleaved memories purpose-built for applications with different patterns of memory access. These examples lay the groundwork for a systematic design process that creates interleaved memories tailored to the specific needs of each different application. We conclude with discussion of FPGA-specific features that can improve performance or reduce resource utilization.

2. MOTIVATING APPLICATIONS

Many compute-intensive applications share a common characteristic: each step in the computation depends on an access cluster of points the grid, close to each other, in fixed positions relative to each other, and always used together in the computation. This broad description covers

* This work was supported in part by NIH award #RR020209-01 and was facilitated by donations from Xilinx Corporation.

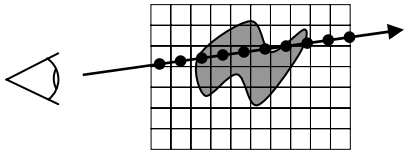

Figure 2. Ray casting: interpolation at off-grid points

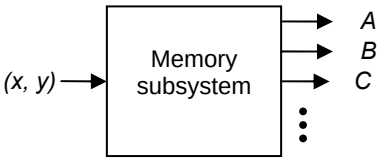

Figure 1. One grid coordinate accesses a cluster of points

many different kinds of problem in one, two, or more dimensions, including:

- *Molecular dynamics*. Particle-mesh models of molecule behavior appear in many applications. They use grids to approximate the vector fields representing forces that act on the atoms. At every time step, interactions between each atom and the force grid define the behavior of the molecule as a whole. Atom positions generally do not align to the grid points at which force values are computed, so 3D interpolation is necessary. Tricubic interpolation using modified B-splines has been found effective [11], and requires 64 grid points in a 4×4×4 neighborhood around each atom.
- *Image processing*. Morphological operators in two [6] and three [9] dimensions have long been staples in image processing, as have convolution kernels of various sizes [5]. In either case, the kernel defines an access cluster for fetching points from the pixel grid being processed. Sparse kernels, containing don't-care patterns or zero coefficients, are common and of special interest. They ignore pixel values in parts of the image region they cover, creating application-specific opportunities for optimization.
- *Volume rendering*. Ray-casting algorithms [13] traverse a 3D grid of values representing the optical characteristics at each point in a 3D field, such as opacity or color in a plume of smoke. Figure 1 shows how a ray samples the field, generally at off-grid points requiring 3D interpolation. Related algorithms have also been found useful in three-axis rotation of volumetric models for *in silico* drug screening [12].
- *Cellular automata*. One-, two-, or three- dimensional grids approximate many kinds of physical phenomena. At each time step, the new state of each grid cell depends on its previous value and on the values of some set of neighboring cells.
- *Red-black relaxation.* Gauss-Seidel relaxation alternates between even- and odd-numbered grid cells. This avoids the need for duplicate memory arrays and avoids numerical problems that arise from computing with incomplete updates to the grid.

These few examples show just a few of the ways in which grid-based computations differ from each other. First, they differ in the number of dimensions of the grid. Second, they differ in the number and position of points in the cluster required at each step. Third, they do not always allow regular or predictable order of access to successive clusters of points. Ray-casting, for example, may traverse the grid along any diagonal. Molecular dynamics, except when dealing with crystals, uses atoms having irregular distributions of position. Some systems for FPGA computation have been able to avoid memory bandwidth problems by heavy reuse of data fetched in tightly controlled sequences [10]. Although effective in some cases, that approach fails for applications that access memory in unpredictable order.

When performing any of these computations, parallelization often requires that multiple points in one cluster be available together, possibly all of them. A von Neumann architecture inherently degrades performance in these cases, requiring that the points be accessed sequentially even when used concurrently. Concurrent access to all of the points, implemented with a structure like that in Figure 2, would give significant performance improvement.

Although different applications have access clusters of different size and organization, each application has a fixed, predictable access cluster. A good FPGA implementation would use the FPGA's many on-chip RAMs for concurrent access to the different points in an access cluster. Ideally, the RAMs would have non-redundant content and high utilization of the bits in each of the RAMs used. Case studies in section 4 show memory structures that meet these goals for a few specific applications. Then, section 5 shows a general approach for designing a memory structure specific to the geometry of any application's access cluster.

3. RELATED WORK

Processor designers have long faced problems caused by CPUs that are able to issue memory requests faster than they can be fulfilled by the memory. Memory interleaving, improves memory bandwidth by dividing memory into separate banks able to process different requests concurrently. As early as 1968, SIMD processors such as the ILLIAC IV [1] used interleaving to support reading of multiple memory words in one access cycle. Other implementations, such as those in the CDC 6400 [4], accessed only one word per cycle. Each memory bank required multiple cycles for reading one word, however, so this allowed concurrency by overlapping several multi-cycle access latencies. Whether the implementation used *broad parallelism* as in the ILLIAC, *pipelined parallelism* as in the CDC 6400, or a combination, high performance

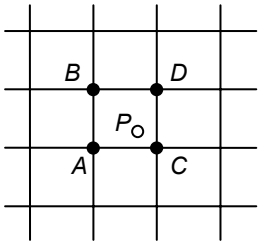

Figure 4. Cluster of points accessed to compute *P* using bilinear interpolation

depended on concurrent activity in multiple memory banks. Pipelined parallelism has historically been more widespread than broad parallelism, because it is compatible with systems that have only one memory access bus. Broad parallelism demands multiple paths from memory to the processing elements, implying high hardware cost when implemented in traditional technologies.

Successful parallelism implied that no concurrent memory accesses would collide with each other by requiring service from the same bank of memory [3]. Many clever techniques have been proposed for reducing the probability of collisions between concurrent memory accesses, such as prime numbers of memory banks [10] or pseudorandom mapping of memory addresses to memory banks [14]. These approaches all implement a fixed structure, without respect to any particular application. They also have the intent of breaking up regularities in the application's data access pattern, so as to reduce the probability of delay-causing collisions at any one bank of memory

Modern FPGAs support different premises for the design of high-performance memory systems:

1. They impose essentially no cost for high-order memory interleaving, since FPGAs already have hundreds of independent RAM banks on chip.
2. For the same reason, current FPGAs impose essentially no cost for broad parallel access to their internal RAMs, as long as fetched data is used by on-chip processing elements. The many memory ports and connectivity resources required for broad parallelism already exist in the FPGA fabric, waiting to be exploited.
3. FPGAs are inherently configurable, so there is no hardware cost in creating memory structures unique to each application, something impossible for any fixed architecture.
4. Most importantly, an FPGA can implement memory structures in terms of the application's indexing strategy and dimensionality, not just in terms of an arbitrary set of memory addresses.

4. CASE STUDIES

These examples show just a few kinds of interleaved memories, demonstrating access clusters of different dimension, size, and geometric character. In each case, the goal is to create a memory subsystem that accepts one grid index as input, and outputs a cluster of grid points *(A, B, C, …)*, as illustrated in Figure 2.

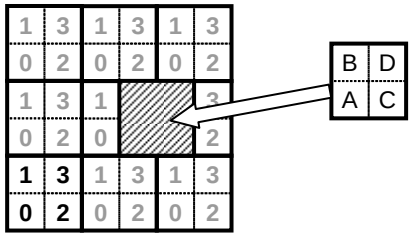

Figure 3. Numbered RAMs tiling the computation grid, and access cluster *ABCD* within the tiling.

4.1. Bilinear interpolation

Error! Reference source not found. shows a point *P* misaligned to 2D grid. The data value at that point is to be computed using bilinear interpolation used the cluster of points *(A, B, C, D)*. The interpolation itself is not of interest at present, only access to the cluster of points needed at that step.

The access cluster consists of points at coordinates *(x, y)*, *(x+1, y)*, *(x, y+1)*, and *(x+1, y+1)*, where point A is considered the origin of the cluster. It is clear that the *x* and *x+1* index pair includes one odd and one even address, and likewise for the y coordinates. Suppose that four RAM banks[1] are available, numbered 0 to 3. Assign each grid point to a RAM bank according to the LSBs of the coordinates, as shown in **Error! Reference source not found.**.

Table 1. Mapping grid points to RAM banks

x LSB	*y LSB*	*RAM bank*
0	0	0
0	1	1
1	0	2
1	1	3

At any one cycle, the four RAM banks can each produce one value needed for an access cluster, so the entire cluster can be accessed at one time. Two problems remain: generating the addresses for each of the RAM banks and aligning the RAM output into some fixed order for presentation to the nest stage of the computation.

In order to understand these two problems, it is helpful to treat the RAM array as a tile that covers the entire grid, as shown in Figure 4. The position of each RAM within the tile is fixed, but there is a different correspondence between the set of RAM banks and the *(A, B, C, D)* tuple according to the position of the access cluster relative to the tile.

[1] RAM banks are logical, not physical structures, implemented using as many of the FPGA's block RAM resources as needed.

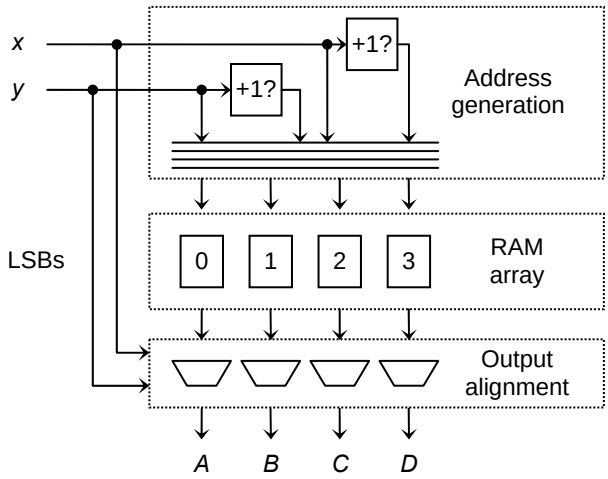

Figure 5. Interleaved memory for broad parallelism

Figure 4 shows the positions of each RAM bank within the basic tile, and the image of each RAM bank in the tiles that cover the grid. This shows an example where access cluster *ABCD* has its lower-left corner at grid coordinate (3,2), assuming 0-based indexing. The cluster aligns to the tile border in the *y* direction, but straddles the tile border in the *x* direction, so that *A* corresponds to RAM bank 2, *B* to bank 3, *C* to bank 0, and *D* to bank 1. For each value within the cluster, the RAM address depends on which instance of the basic tile it touches, is may differ if (as here) the cluster crosses tile boundaries. The mapping between RAM banks and cluster values depends on alignment between the cluster and the tile pattern.

For convenience, the grid coordinate of the access cluster as a whole is taken to be the coordinate of its lower-left corner. After extracting the least significant bits (LSBs) from the cluster's *x* and *y* coordinates, two-dimensional array indexing is converted into a linear memory address in the usual way:

$$Addr = N_Y * x' + y' \qquad \textbf{Equation 1.}$$

where *Addr* is the memory address, N_Y is the size of the *Y* dimension of the array, and *x'* and *y'* are the indices after removal one LSB from each. For convenience, this memory address is written as *[x', y']*, where square brackets distinguish the memory address from a grid coordinate.

The cluster accesses indices *x* and *x+1*. That requires access to memory locations based on *x'* and *(x+1)'*, i.e. with the LSBs removed. If the original *x* is even, then *(x+1)' = x'*. If *x* is odd, then *(x+1)' = x'+1*. Addresses for RAMs 2 and 3 use *x'*; addresses for RAMs 0 and 1 use *(x+1)'* which may have a different value. The same reasoning applies to *y* indices, but for RAM pairs 0,2 and 1,3. The address for each RAM, then is given by Table 2.

For ease of implementation, the computation section fed by this memory array assumes that the four values are presented in fixed order: *A, B, C, D*. If *(x, y)* is (even, even), then RAM 0 generates A. If (x, y) is (even, odd), then the A output comes from RAM 1. Likewise, RAM 2 supplies A for (odd, even) values and RAM 3 supplies A for (odd, odd). The following table shows which RAM provides each of the four outputs according to the even/odd states of the indices:

Table 2. RAM addresses for bilinear interleaving

RAM bank	*Address*
0	[if *x* even then *x'* else *x'+1*, if *y* even then *y'* else *y'+1*]
1	[if *x* even then *x'* else *x'+1, y'*]
2	[*x'*, if *y* even then *y'* else *y'+1*]
3	[*x', y'*]

Table 3. RAM to output mapping by *(x,y)* LSBs

Out	*even,even*	*even,odd*	*odd,even*	*odd,odd*
A	0	1	2	3
B	1	0	3	2
C	2	3	0	1
D	3	2	1	0

Each row of this table can be written as a multiplexer (or mux). The mux's output is one of the points in the cluster, *A*, *B*, *C*, or *D*. Data inputs to the mux come from one of the RAMs, 0, 1, 2, or 3. Select control for the multiplexer comes from the combination of *x* and *y* LSBs.

At this point, the design of the interleaved memory array is complete. Figure 5 shows its internal structure. The subsystem is indexed using the *(x, y)* coordinate that represents the position of the access cluster. The blocks labeled *+1?* in the *Address generation* section perform the conditional increment operations of Table 2. The resulting index values are combined in various ways to generate the addresses to the RAM banks in the RAM array section. Each RAM bank generates its output, which is passed to the *Output alignment* section where the logic of Table 3 is implemented.

This over-all structure is used in every interleaved memory system described in this paper. When a new interleaving is created for a different application, only the following features differ:

- *Index variables*. Figure 3 shows *(x,y)* grid indexing for a 2D computation. Problems of 1, 3, or other dimension have different numbers of indices.
- *LSBs*. In this example, one LSB is extracted from each index variable. Other applications extract two or more LSBs, and some extract different numbers of LSBs from different indices.
- *Address generation*. This application conditionally increments index values according to odd or even index values. Other applications conditionally increment according to other tests, and combine grid coordinates into memory addresses in different ways.

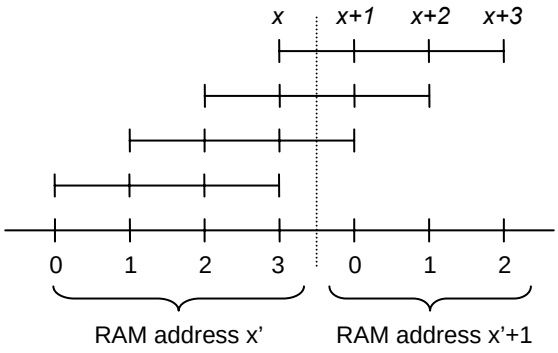

Figure 7. RAM addresses: *x'* or *x'+1* depending on initial offset

- *RAM array*. The number of RAMs depends on the interleaving required for the particular application.
- *Output multiplexing*. This has one output per element of the access cluster, a number that differs for different applications. The mapping of RAMs to outputs also varies, according to application-specific uses of the index LSBs.

In addition, the number of bits in the index values, RAM addresses, and data words may also vary between applications.

4.2. Tricubic interpolation

The motivating application for this case study comes from molecular dynamics [11]. It represents potential fields acting on atoms as values stored in a grid, with 3D interpolation used to determine the field values at off-grid points representing atoms. The authors observe that cubic interpolation gives better results than linear interpolation. Instead of a 2×2×2 access cluster of grid points around each atom, however, tricubic interpolation requires a 4×4×4 set of grid points.

This extends the first case study, bilinear interpolation, in two directions. First, it uses three index values *(x, y, z)* rather than two. Second, it requires four points in each axis for finding the cubic equation to interpolate, rather than the two needed for linear interpolation. Again, the interpolation arithmetic is not of interest for purposes of current discussion. We describe only the interleaved memory needed for fetching the full set of 4×4×4=64 values in each access cluster.

Since there are 64 values in each access cluster, interleaving requires 64 RAMs to ensure non-interfering access to all values in the cluster. The cluster is a 4×4×4 cube, with the same indexing behavior in the *x*, *y*, and *z* axes. We examine the *x* axis first, then treat the other axes in analogous ways.

Each access cluster fetches values from indices *x*, *x+1*, *x+2*, and *x+3*. Examine the two LSBs: each possible value

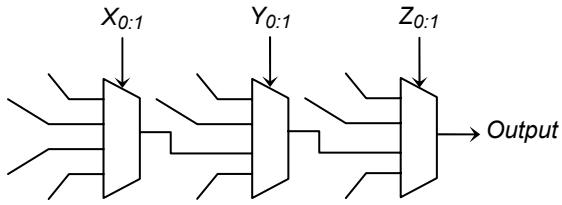

Figure 6. 64:1 mux, axis by axis

(0, 1, 2, and 3) occurs once in this group, so those LSBs will be used for output multiplexing. The *x'* values are *x* index values stripped of their LSBs, as in the bilinear case, but this application strips two LSBs rather than one.

Considered modulo 4, there are four possible starting points for the access cluster, shown in Figure 6. If the cluster starts at a 0 mod 4 boundary, then all four *x'* values are the same. If the cluster starts at a 1 mod 4 boundary, then *(x+3)'=x'+1* for RAM bank 0, and so on for starting offsets 2 and 3 mod 4.

Table 4 corresponds to Table 2, but shows one index only. The full form of Table 4 has 64 entries, one for each of the RAMs, and handles cases for all combinations of *x, y* and z offsets mod 4.

Table 4. *x'* address increments by *x* offset**.**

RAM	*Address*
0	[if *x mod 4* ≥ *1* then *x'+1* else *x'*]
1	[if *x mod 4* ≥ *2* then *x'+1* else *x'*]
2	[if *x mod 4* = *3* then *x'+1* else *x'*]
3	[*x'*]

The *Output alignment* section of Figure 5 is also implemented in this application. This time, however, *Output alignment* has 64 data inputs, 64 outputs, and 6 bits of selection control, two LSBs from each of the three indices.

A naïve implementation would built the section in terms of 64-input muxes. Straightforward synthesis generates a huge amount of logic for that mux, and repeats it for each bit in each of the 64 output words. The implementation shown in Figure 7 takes advantage of application-specific knowledge of the selection value, i.e. that it is composed of three two-bit fields. When synthesized for Xilinx FPGAs, this cascade uses six slices per data bit, instead of 32 for the 64-input mux.

This application demonstrates the value of FPGAs in creating interleaved memory structures. First, the FPGAs like the Xilinx Virtex family easily satisfy the need for 64 independently accessible memory busses. The VP100 chips in that family have over 400 RAMs available, but even 64 memory busses would be expensive in competing technologies. Second, even if data words are only 16 bits each, every cluster access fetches 64*16=1024 bits of data

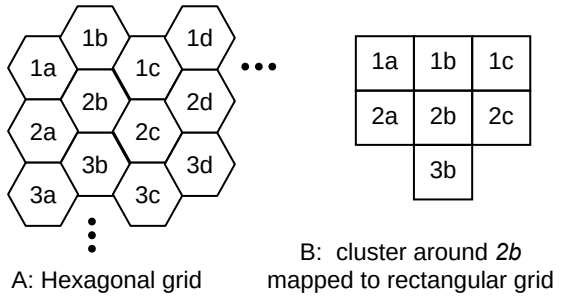

Figure 8. Hexagonal access cluster mapped to rectangular grid

– a 1Kbit transfer. Again, the FPGA's connectivity resources can handle this word size readily, but it would be problematic to handle in many other implementation technologies.

4.3. Hexagonal grids

Hexagonal grids rather than rectangular ones arise in some problems. The following diagram show an access cluster within a hexagonal grid:

This presents two problems. The first is that the grid must be mapped to a rectangular structure to simplify 2D array indexing. That problem is solved in Figure 8B, which shows the cluster centered on grid cell 2b. The second problem is that the access cluster is not a power of two in size. Instead, its bounding box is 3×3. It is possible to perform mod-3 and divide-by-three arithmetic on the grid indices, if the logic delays in addressing are acceptable.

Another way to think of this cluster is as a 7-element subset of 4×4 RAM array. This makes address computation easy, but require 16 RAM banks for accessing a cluster of size 7. The additional RAMs may, at first glance, look like 130% overhead in the RAM allocation. On the other hand, adding those RAMs reduces the amount of addressing logic, possibly including block multipliers, so reduces the delay in the addressing logic path. Also, depending on the total capacity of the RAM array, those RAM resources could have been required anyway, in order to provide the total number of words required for the memory structure. It is also worth noting that the Xilinx VP100 contains over 400 RAMs. Resource tradeoffs differ between applications, but this can be an effective use of additional RAM to reduce logic delays and reduce consumption of other FPGA resources.

Assuming the 4×4 RAM array, implementation is now straightforward. Address generation logic resembles that used in the tricubic interpolation example, but simplified to two dimensions. The *Output alignment* section has 16 data inputs, 7 data outputs, and 4 selection bits (two LSBs from *x* and two from *y*).

5. DESIGNING THE INTERLEAVED STRUCTURE

Design of the interleaved memory proceeds in a regular sequence of steps:

1. Define the access cluster used by the application. In cases like the hexagonal grid example, this may require some effort in converting the grid to a rectangular form.
2. Round the cluster size up to the next power of two in each dimension. Allocate one RAM bank per rounded-up cluster element to the RAM array of Figure 5.
3. Create the *Address generation* network following the examples leading to Table 2 and Table 4.
4. Create an output mapping table like Table 3, with one output per element in the actual access cluster (not rounded up). Use that table to create the *Output alignment* section of the array.

Step 1 requires insight into the application's unique pattern of memory reference. This may involve tradeoff decisions if the FPGA implementation requires partitioning the application in ways that affect memory access. As in the hexagonal grid example, additional insight may be needed for mapping the application to a rectangular grid. After that, generation of the interleaved memory system follows in a straightforward way.

This approach can be modified in many ways, to make better use of available resources or to take advantage of features unique to a particular model of FPGA.

5.1. Multiported RAM

Block RAMs in many FPGA families have two independently addressable read ports. These can be used, in some designs, to reduce the number of RAMs needed for the RAM array of Figure 6.

Consider the array of four RAM banks used in the bilinear interpolation case study of section 4. The bank number for each of the four RAMs in that example is constructed from the LSBs of the *x* and *y* grid coordinates, as shown in Table 1.

This can also be implemented with two dual ported RAMs, named 01 and 23. RAM addresses *[x', y']* are built from the *x* and *y* values stripped of their LSBs, as before. The difference is that RAM 01 holds grid points for all points with even *y* coordinates and RAM 23 holds all points that have odd *y* coordinates. Then treat port 0 of RAM 01 the same way as RAM 0 from the original example, RAM 01 port 1 as RAM 1, and so on.

5.2. Broad parallel writing

The discussion so far has assumed that grid computations are dominated by read access to stored grid values. That is not true in all phases of grid-based computations, however. Figure 9 shows the cycle that occurs in particle-mesh (PM) molecular dynamics models.

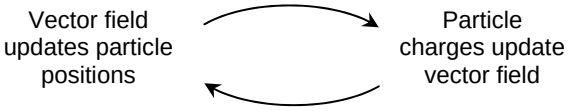

Figure 9. Particle-mesh calculation cycle (after [8])

One phase of the calculation updates the grid of Coulombic potential values by summing the spatial distributions of each particle's charge [8]. Each particle's charge affects a large spatial region, so must update many points in the grid. In order to use broad parallelism at this step, multiple grid points must be modified. In some implementations, this requires a second multiplexing network similar to the *Output alignment* of Figure 5, but arranged to map data input clusters to RAM banks rather than RAM to output.

5.3. Non-2^N RAM arrays

The case studies of section 4 refer to the LSBs of grid coordinates and to coordinate values stripped of their LSBs. These, of course, are optimizations of $x \bmod k$ or $\lfloor x/k \rfloor$, where k is some power of two. That is a convenience only, not a fundamental restriction on the values that may be used for the RAM array. With some additional complexity in the address generation logic, other integer sizes of RAM array can be accommodated.

For example, the case study on hexagonal grids defines an access cluster that fits a 3×3 bounding box, requiring 9 RAM banks. The RAM array was rounded up to 4×4, requiring 16 RAM banks, potentially an increase of 78% in RAM allocation. As noted earlier, the additional RAM banks represent an addition of RAM hardware that reduces access times and address generation logic. Different time and resource tradeoffs might favor the following solution.

In building the memory for a 3×3 RAM array, the addressing logic of Equation 1 has to be interpreted slightly differently. The N_Y value becomes 3, x' becomes $\lfloor x/3 \rfloor$, and y' becomes $\lfloor y/3 \rfloor$. Direct division of $\lfloor x/3 \rfloor$ in digital logic is inconvenient. Instead, the same net effect comes from multiplying x by $^1/_3$ in fixed-point format, i.e. $0.0101\ldots_2$, where enough bits of precision are maintained to make $x \times {}^1/_3$ exactly $\lfloor x/3 \rfloor$ over the entire range of x values used by the application. One 18×18 block multiplier can handle $x \times {}^1/_3$ with enough accuracy to cover a 17-bit range of coordinates, and multipliers can be ganged for wider index ranges.

The LSBs of x and y in fact represent $x \bmod N_X$ and $y \bmod N_Y$, for RAM array size $N_X \times N_Y$. In the case of the 3×3 RAM array for the hexagonal grid, the mod 3 residue is required in both coordinate values. Like division, the arithmetic modulus can be computed with a modest amount of logic.

Start with the observation that

$$(a + b) \bmod 3 = ((a \bmod 3) + (b \bmod 3)) \bmod 3,$$

which extends recursively to any number of addends. Take the binary representation of x to be $\ldots x_8\, x_4\, x_2\, x_1$, where x_J is the bit of weight J in positional notation. Then $x \bmod 3 =$

$$\begin{aligned} & x_1 \bmod 3 && = 0 \text{ or } 1 \\ + & x_2 \bmod 3 && = 0 \text{ or } 2 \\ + & x_4 \bmod 3 && = 0 \text{ or } 1 \\ + & x_8 \bmod 3 && = 0 \text{ or } 2 \\ + & \ldots && \ldots \end{aligned}$$

Coefficients 1 and 2 are the possible values of $2^N \bmod 3$, and other coefficients would appear if the RAM array dimension were some value other than 3. The sum computed in this way grows only slowly: it is never more than 15 for a ten-bit grid coordinate. A small lookup table can readily provide the mod 3 residue of that sum.

Given these changes of interpretation, the directions of section 5 can be used with one modification: step 2 does not round up. The steps in this example are:

1. Define the access cluster. In this case, the 7-point access cluster for the hexagonal grid is used.
2. Take the bounding box (3×3) of the access cluster to be the dimension of the RAM array.
3. Create the *Address generation* network:

RAM	*Address*
00	[if x mod 3 > 0 then ⌊x/3⌋+1 else ⌊x/3⌋, if y mod 3 > 0 then ⌊y/3⌋+1 else ⌊y/3⌋]
01	[if x mod 3 > 0 then ⌊x/3⌋+1 else ⌊x/3⌋, if y mod 3 > 1 then ⌊y/3⌋+1 else ⌊y/3⌋]
02	[if x mod 3 > 0 then ⌊x/3⌋+1 else ⌊x/3⌋, ⌊y/3⌋]
10	[if x mod 3 > 1 then ⌊x/3⌋+1 else ⌊x/3⌋, if y mod 3 > 0 then ⌊y/3⌋+1 else ⌊y/3⌋]
11	[if x mod 3 > 1 then ⌊x/3⌋+1 else ⌊x/3⌋, if y mod 3 > 1 then ⌊y/3⌋+1 else ⌊y/3⌋]
12	[if x mod 3 > 1 then ⌊x/3⌋+1 else ⌊x/3⌋, ⌊y/3⌋]
20	[⌊x/3⌋, if y mod 3 > 0 then ⌊y/3⌋+1 else ⌊y/3⌋]
21	[⌊x/3⌋, if y mod 3 > 1 then ⌊y/3⌋+1 else ⌊y/3⌋]
22	[⌊x/3⌋, ⌊y/3⌋]

4. Create the *Output alignment* network. Construct a table (not shown) where each row represents one of the 7 outputs. Each column represents one of the 9 possible offsets between output cluster and RAM array, specified by a different *x mod 3*, *y mod 3* pair. Each table element is the RAM bank that provides data to that output at that offset. Implement each row of the table as a 9:1 mux generating one of the cluster values.

6. SUMMARY AND FUTURE DIRECTIONS

Memory interleaving is a well-known technique for improving the bandwidth of a memory system by increasing parallelism. Pipelined parallelism has traditionally been more common than broad parallelism because it requires less interconnection hardware for

implementation. Whether for pipelined or broad parallelism, standard memory interleaving is based on the memory addresses only, and application knowledge consists of broad assumptions about the behavior of many dissimilar applications.

This paper presents a widely applicable technique for creating interleaved memory structures that take advantage of the unique strengths of FPGAs. The technique creates memory interleaving structures that offer broad parallel access to the clusters of data used by an application, based on knowledge of the application's memory reference patterns. These memories store data without replicated storage of data values and without wasted memory locations, ensuring efficient use of FPGA resources. We are currently developing tools for automatic creation of these interleaved memory structures.

Extensions to this basic scheme are possible. In particular, there are applications in which a grid is used multiple ways, e.g. is updated, then is used in a force field calculation, as in molecular mechanics applications. It is not necessarily the case that both kinds of memory reference use the same access clusters. Elaborations of the techniques shown in section 5 may be able to reuse one memory array with addressing and alignment sections that suit access clusters of different characteristics. We are also exploring additional optimizations that appear to be possible with some sparse access clusters, where RAM array sizes may be reduced by taking advantage of the holes in the cluster.

Memory interleaving for FPGA-based computing is also effective for improving memory bandwidth and parallelism in the computation stages. FPGA computing differs from traditional computing in a few critical ways. First, FPGAs offer massive on-chip memory parallelism at essentially no cost. Second, because they are configurable, they can implement interleaving strategies tuned to the specifics of each application, sometime more than one strategy in different parts of the circuit. Third, FPGAs support the wide data words, 1Kbit or more, needed for access clusters covering dozens of grid points. Fourth, because every FPGA computation is an application-specific circuit design, the application's logical indexing is accessible to the memory designer, not just its sequence of address references. These features, memory and connection resources, configurability, and application knowledge, create unique opportunities for optimizing the FPGA's memory implementation to the application at hand. This paper presents a step by step technique for creating such memories, applicable to many families of compute intensive applications, not just volume rendering, image processing, and computational chemistry.

7. REFERENCES

[1] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes. "The Illiac IV Computer." *IEEE Trans Computers* 17(8), Aug 1968

[2] A.P.W. Böhm, B. Draper, W. Najjar, J. Hammes, R. Rinker, M. Chawathe, and C. Ross. "One-step Compilation of Image Processing Applications to FPGAs". *IEEE Symposium on Field-Programmable Custom Computing. 2001*

[3] P. Budnik and D. J. Kuck. "The Organization of Parallel Memories" *IEEE Trans Computers* 20(12)1566-1578, Dec 1971

[4] Control Data Corporation. "Control Data 6400/6500/6600 Computer Systems Reference Manual." 1969

[5] B. K. Davies. "Machine Vision: Theory, Algorithms, and Practicalities." Morgan Kaufmann. 2005.

[6] E. R. Dougherty. "An Introduction to Morphological Image Processing." SPIE Optical Engineering Press. 1992

[7] D. S. Ebert and R. E. Parent. "Rendering and animation of gaseous phenomena by combining fast volume and scanline A-buffer techniques". *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques SIGGRAPH '90*

[8] P. Gibbon and G. Sutmann. "Long-Range Interactions in Many-Particle Systems." In Quantum Simulations of Many-Body Systems: From Theory to Algorithms. NIC Series 10:467-506. 2002

[9] P. P. Jonker. "Morphological Image Processing: Architecture and VLSI Design." Kluwer Technische Boeken B. V. 1992

[10] D. H. Lawrie. "Access and Alignment of Data in an Array Processor". *IEEE Trans. Computers* 24(12), Dec 1975.

[11] D. Oberlin, Jr. and H. A. Scheraga. "B-Spline Method for Energy Minimization in Grid-Based Molecular Mechanics Calculations". *J. Computational Chemistry* 19(1)71-85. 1998.

[12] T. VanCourt, Y. Gu, V. Mundada, and M. Herbordt. "Rigid Molecule Docking: FPGA Reconfiguration for Alternative Force Laws." European Journal of Applied Signal Processing, to appear 2006

[13] A. Watt and M. Watt. "Advanced Animation and Rendering Technique." Addison Wesley. 1992

[14] S. Weis. "An aperiodic storage scheme to reduce memory conflicts in vector processors." *Proc. International Symposium on Computer Architecture*. 1989