A Runtime Framework for Optimizing Multi-Dimensional Array Accesses on Multi-core Processors

Jae-Seung Yeom and Dimitrios S. Nikolopoulos*
Department of Computer Science
Virginia Tech
Blacksburg, VA, USA
{jyeom, dsn}@cs.vt.edu

Abstract

Scientific and numerical applications rely on multi-dimensional array data accessed in nested loops. These regular data access patterns can benefit from explicitly managed local memories, such as the local stores of the Cell processor. We present Strider, a runtime library framework which helps programming and optimization of multi-dimensional data accesses in nested loops, on multi-core processors with explicitly managed memory hierarchies. Strider automatically schedules strided data accesses based on a high-level description of loops and arrays provided by programmers. We present the design and implementation of Strider on Cell, followed by a preliminary performance evaluation of the framework. Our evaluation illustrates that Strider outperforms by significant margins existing parallel programming environments for multi-core processors with explicitly managed memory hierarchies.

1. Introduction

Scientific and engineering applications rely heavily on multi-dimensional array data accessed in nested loops. Such applications with regular data access patterns can take advantage of an explicitly managed memory hierarchy (EMMH), such as the local stores of the Cell Broadband Engine (Cell BE), by prefetching long chains of data to hide memory latency.

Several programming frameworks have been proposed to hide the complexity of managing EMMHs from programmers [1–6]. They are categorized into runtime library frameworks and code generation frameworks with high-level parallel programming models [1].

Library frameworks provide low-level interfaces for describing strided array accesses (such as data transfer lists in the case of IBM’s Accelerated Library Framework [2]), but leave the responsibility of efficiently using these interfaces to the programmer.

Code generation frameworks rely on compiler directives or programming language constructs. Some code generation frameworks such as Sequoia and CellSs naturally support strided accesses in nested loops [3, 4]. However, the optimization abilities of these frameworks are often limited, either due to limitations of static analysis or due to limitations in the overall expressiveness of the compiler directives or language constructs used in these frameworks. To statically optimize strided data accesses in nested loops in the aforementioned frameworks, the number of loop levels has to be fixed and known at compile time, the iteration space should not be divided across multiple levels of a nested loop, the maximum depth of a nested loop needs to be limited, and the maximum number of dimensions of the arrays accessed in the loop needs to be limited. Syntactic language limitations may further prevent optimization.

An additional practical issue is the limited size of local stores in EMMHs. On Cell BE for example, the local store of each SPE (a vector SIMD core carrying the bulk of operations in data-parallel computations) is limited to 256 KB for both code and data. Many code generation frameworks tend to generate customized and optimized codes for each task and these codes may easily fill up local stores. From this point of view, a library framework is preferable, as its code can be optimized and then reused across tasks.

We present a new framework, Strider, which is an effort to overcome the limitations of code generation frameworks and library frameworks in handling strided array accesses in nested loops running on multi-core processors with EMMH. Strider is a library framework which may be used directly by programmers, or as a runtime system during code generation by compilers. We focus on the use of Strider as a library framework by programmers in this paper and present its design, im-

*Also with the Institute of Computer Science, Foundation of Research and Technology – Hellas (FORTH-ICS).
plementation, and evaluation on Cell. Strider demonstrates good scalability, high utilization of on-chip bandwidth and higher overall performance (up to 51% with the copy and transpose benchmarks discussed in Section 5) than Sequoia, a mature language-based parallel programming framework for EMMHs.

The rest of the paper is organized as follows. Section 2 introduces the architecture of Strider. Section 3 discusses the design and implementation of Strider. A simple code example using Strider is presented in Section 4. Preliminary experimental results are presented in Section 5. Section 6 discusses future work. Section 7 concludes the paper.

2. Overview of Strider

Strider comprises a runtime system, user-defined modules and a programming API. The runtime system has seven components and there are three types of user-defined modules as shown in Figure 1. The framework is implemented on Cell, a heterogeneous multi-core processor featuring a PowerPC SMT core (PPE) and eight vector-RISC cores (SPEs).

The programmer sets parameters for the runtime system and defines the computation to be performed on data in the user-defined modules via the provided API. The runtime system manages data transfers. Figure 1 summarizes the structure of the runtime framework. The white rectangles with rounded corners are the runtime components provided by Strider. The shaded rectangles are what programmers write conforming to Strider specifications and using the Strider API. The arrows connecting modules represent the flow of module attachments.

The block access list generator is the key component which is unique in Strider. It generates DMA lists based on the description of the iteration space of a nested loop and those of the arrays accessed in the loop. Such information is provided in the SPE offloaders via the Strider API. The DMA handler provides the abstracted interfaces to the DMA commands of IBM Cell SDK. The handler makes DMA calls based on the information provided by the internal block access list structure generated by the block access list generator, as well as information provided by the buffer manager such as the address of the local buffer and the DMA tag for the buffer.

Programmers can plug their computation kernels into the framework via the user-provided kernel interface at runtime. The buffer manager allocates data buffers in the local store, schedules DMA list generation and DMA transfers to move data into and out of the buffers and finally invokes the user-provided kernel.

From the programmer’s perspective, calling the buffer manager in the SPE offloaders with the description of an iteration space and data arrays is sufficient to run the code with automated computation-communication overlap. DMA list generation, DMA communication and kernel execution are scheduled by the runtime system in a way that communication is overlapped with computation via multi-buffering. Data known to be used by future computation is pre-loaded into cyclically allocated buffers. For arrays that are either read or written in a computation kernel, Strider uses double-buffering. For arrays that are both read and written in a computation kernel, Strider uses triple-buffering.

PPE and SPEs use signals for data transfer completion notification and pass runtime parameters via the synchronization framework. The PPE offloader handles signaling and parameter passing to SPEs by directly writing into or reading from the effective addresses of local store variables shared between the PPE and each SPE, thus relying on on-chip communication. The debugging framework is optional. It provides detailed execution information, as well as details on generated exceptions, if any. The profiling framework is also optional and provides breakdowns of the cost of computation and data transfers for performance tuning.

3. Design and implementation of Strider

3.1. Representing arrays and loops

Strider requires the programmer to describe the iteration space of nested loops and the arrays accessed in the loops. From this information, the runtime system identifies the access pattern and schedules accesses using direct buffering in local stores. We present the representation of iteration spaces and arrays in this section. We introduce the key data structures to help understanding the underlying mechanisms of the framework.
Table 1: The description of array double A[nx][ny][nz][5]

<table>
<thead>
<tr>
<th>number of dimensions</th>
<th>dim 4 size</th>
<th>dim 3 size</th>
<th>dim 2 size</th>
<th>dim 1 size</th>
<th>element size</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>nx</td>
<td>ny</td>
<td>nz</td>
<td>6 (=3+pad)</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 2: The loop parameters for the loop shown in Figure 2

<table>
<thead>
<tr>
<th>induction variable</th>
<th>array dimension</th>
<th>start</th>
<th>end</th>
<th>step size</th>
<th>loop level</th>
<th>step direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>4</td>
<td>0</td>
<td>nx</td>
<td>3</td>
<td>4</td>
<td>forward</td>
</tr>
<tr>
<td>j</td>
<td>3</td>
<td>1</td>
<td>ny-1</td>
<td>2</td>
<td>3</td>
<td>backward</td>
</tr>
<tr>
<td>k</td>
<td>2</td>
<td>1</td>
<td>nz-1</td>
<td>1</td>
<td>2</td>
<td>forward</td>
</tr>
<tr>
<td>m</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>forward</td>
</tr>
</tbody>
</table>

for (i = 0; i < nx; i += 3) {
  for (j = ny-2; j >= 1; j-=2) {
    for (k = 1; k < nz-1; k++) {
      for (m = 0; m < 5; m++) {
        C[i][j][k][m] = d*A[i][j][k][m]+B[i][j][k][m];
      }
    }
  }
}

Figure 2: An example of a nested loop

3.1.1. Array descriptor. Each processor with an explicitly managed local store in Strider needs to be provided with information on the arrays to be transferred to its local store. An array is described by the data structure array descriptor. The structure carries the effective address of an array, the number of dimensions, the size of each dimension and the size of an array element, which is treated as the size of dimension 0. For example, for the array defined as double A[nx][ny][nz][5], the size of the fourth dimension is nx and the size of the third dimension is ny as shown in Table 1. In the implementation of Strider on the Cell, the array descriptor is set on the PPE for a statically defined array and transferred to each SPE once at the beginning of execution. For a dynamically allocated array, the array descriptor is set and transferred to SPEs after the array is allocated and before it is referenced by SPEs.

3.1.2. Array padding. Strided accesses can be complicated due to imposed alignment constraints for DMAs [7]. In Strider, the first dimension of an array is padded to a 16-byte address boundary, such that the position of data at the beginning of any higher dimension is also aligned to such a boundary. The underlying DMA engine requires such an alignment to transfer data larger than or equal to 16 bytes. The same alignment works for transferring naturally aligned data of smaller sizes. The programmer needs to follow this convention by relying on the provided API to define an array statically or allocate an array dynamically. To allocate an array dynamically, the programmer calls an allocation function provided by the Strider API. Strider hides the details of padding and alignment for dynamically allocated arrays. However, when an array A is defined statically, it has to be modified to conform to Strider’s alignment conventions. For example, the array used in the example of Section 3.1.1 needs to be declared statically as double A[nx][ny][nz][5+padding(double,5)] _align.

In general, it is challenging to hide alignment details from the programmer even with code generation, because the array may also be accessed in the non-offloaded region of the application code, which is handled outside of the Strider framework. Changes in the array layout may also require changes in the code that references array elements. An alternative chosen by the Sequoia programming language is to maintain two copies of an array, one for the framework and one for the region outside of the framework [3]. The runtime system copies data between the two arrays when the program switches between two regions. Unfortunately, copying may severely degrade performance.

Strider supports an optional data padding mechanism, which pads each element of an array such that the size of an element is aligned to the 16-byte alignment boundary. This feature is used as a last resort, as it requires large additional memory space especially with native data types such as int, char, and double. Instead, the strategy preferred by Strider is to group data and transfer them in bulk while satisfying the alignment requirements of the DMA engine whenever possible.

3.1.3. Iteration space. The data structure IterSpace defines the access profile of a loop which maps an iteration \( i \in I \) for a given array \( A \) to a physical memory offset \( o \in O \) corresponding to the array element accessed in the iteration as:

\[
IterSpace_A : I \rightarrow O
\]

\( I \) is determined from loop parameters, namely the bounds and step for each level of the loop. Table 2 shows the loop parameters for the loop shown in Figure 2.

The loop parameters are set for each loop level via the data structure IterBasis. This data structure also keeps parameter values in a way that they can be referred to by other modules with the least amount of computation and conditional branches. \( O \) is determined from the array description and the array index term.

For each \( o \in O \), the effective address \( ea \) of the memory location is obtained by adding the beginning effective address of the array \( A \) being accessed, \( ea_A \) :

\[
ea_i = ea_A + o_i
\]

where \( o_i \) and \( ea_i \) are the offset and the effective address at iteration \( i \) respectively.

Calculating offsets of array elements can be prohibitively expensive on Cell. An integer multiplication (7 cycles) is particularly expensive compared to an integer addition (2 cycles) or a load operation (6 cycles) on SPE. The calculation of \( o \) is optimized with an Iterator data structure, which calculates the current effective address by adding a fixed stride to the last effective address used, instead of relying on multiplications with the loop induction variable while looping through the itera-
amplitude into the fundamental monochromatic and the harmonics of a monochromatic. Monochromatics are categorized pattern, and to the same monochromatic group. The first wave is 
\[ i < N; \ i \leftarrow i + 1 \] 
and an integer type array \( B[N] \) and a function of iteration – i.e., a wave – and the waves of the same period are grouped as a monochromatic group. Each wave is associated with an effective address and an offset of its own. The offset of wave allows more flexible description of an access pattern and helps grouping similar patterns:

\[ ea_i = ea_A + o_i + offset_A \]  
(3)

This technique allows to treat \( A[i-1] \) and \( A[i+1] \) as the same access pattern requiring the calculation of \( o \) once for both accesses. It also helps simplifying the iteration space representation and optimizing data transfers by allowing data blocking dynamically up to a loop level where the iteration range does not span over the entire array dimension, as discussed later in Section 4. The amplitude of wave handles different element sizes. Among monochromatic waves, the first wave has the smallest element size, and the sizes of the rest are multiples of the size of the first wave. \( o \) is calculated once for the first wave and scaled for the rest. In this way, similar access patterns of different data types are handled together, thus saving on the memory space required to represent the pattern of each type and reducing the cost of calculating \( o \). For example, a double type array \( A[N] \) and an integer type array \( B[N] \) in the loop for \( i < N; \ i \leftarrow i + 1 \) \( A[i] = B[i] \times A[i] \) are defined as waves of the same monochromatic group. The first wave is \( B \) with amplitude 1 and the second is \( A \) with amplitude 2.

In short, a monochromatic group represents an access pattern, and \( o_i \) is calculated only once for all the waves of a monochromatic. Monochromatics are categorized into the fundamental monochromatic and the harmonics of a fundamental as in Fourier series analysis. In the former, accesses are associated with each iteration, while in the latter, the period is larger than one. There can be at most three fundamentals, one for \( i \) (read-only), one for \( o \) (write-only) and one for \( inout \) (read-write) data.

3.1.4. Block transfers. Data blocking in Strider is differentiated from the working set of a computation. A single block transfer may contain multiple working sets, grouped together to satisfy DMA alignment constraints. Block transfers may be grouped further at an arbitrary loop level to reduce communication overhead or to provide SIMD-friendly data layout. However, this optimization is left as future work.

In code generation frameworks, the shape of blocks transferred between levels of the memory hierarchy is often limited by design for the convenience of static analysis. For example, in Sequoia [3], the iteration space is derived from the description of blocks as well as array parameters, and the shape of blocks is limited to the \( n \)-dimensional hypercube (6-dimensional to be precise). In Strider, the shape of a block is represented separately from the iteration space, thus allowing the programmer to choose among various shapes. The programmer decides the shape of blocks first and then determines how to represent data arrays and the loop for accessing the arrays. The unit of transfer of blocks may be different than the size of any array element in the block for optimization purposes. An example of this case is shown in Section 4. An effective address obtained from Equation 3 is the address of the beginning of a block. The shape of a block is defined with a block descriptor. In Strider, it is either an \( n \)-dimensional hypercube or an amorphous shape. The latter shape relies on a template DMA list to which the address offset for the current iteration is added. It is possible to represent a non-hypercube block with a collection of hypercubes. However, even in such a case, the cost of handling the collection may be far greater than that of a single block.

3.1.5. The parameter stack of a user kernel. A user kernel can be written independently of the runtime framework and plugged into it later at runtime dynamically as long as it conforms to the function prototype of the Strider user kernel. A parameter stack of a pre-defined stack is used with a kernel. The parameter stack is divided into two parts, the internal parameter stack and the user parameter stack. The internal parameter stack is used by the runtime framework to pass internal information to a kernel. This internal information includes the current iteration indices, padding size, alignment size, block size, buffer size, and the number of waves. User parameters can be passed to a kernel via push, pop, set and peek operations. The last two are the methods used to directly access the user parameter stack, such that parameters can be read or modified without being popped
3.2. Decomposition strategy

To divide and distribute an iteration space, Strider offers two decomposition methods. The first method is to divide the entire iteration space of a nested loop considering it as a homogeneous iteration space. This approach often results in better load balancing. However, it is not always possible to consider the entire iteration space homogeneous for various reasons. The most common is when there is a context of computation that has to be maintained through iterations at a certain loop level. The second method allows specifying the range of loop levels where the decomposition needs to be applied. In this approach, nested loop levels are divided in three layers, the upper layer, the middle layer (called decomposition layer), and the lower layer (called streaming layer). The data accessed in the iterations within the same lower layer is delivered to the same SPE. The iteration space within the decomposition layer is divided and distributed as evenly as possible across SPEs. The upper layer is not decomposed and not handled by the SPE. To execute the entire iteration space in the upper layer, the PPE must initiate the task of the other two layers sequentially.

4. An example of coding with Strider

We use the nested loop in Figure 2 as an example of how Strider is used by a programmer. The loop has no loop-carried dependencies. The offloaded loop is composed of two pieces of codes, the SPE offloader shown in Figure 3 (naïve version) and 4 (optimized version), and the user kernel shown in Figure 5. The differences between the naïve and the optimized versions will be explained later in this section.

The SPE offloader code shown in Figure 3 defines the shape of transferred blocks, describes the iteration space of the offloaded loop, sets the user-provided kernel, and calls the buffer manager. Lines 7–10 declare the structures used in the code. The set_blockdesc() interface defines the shape of a transferred block in line 12. As explained in Section 3.1.3, the mapping between the iteration space and memory locations is defined by the IterSpace structure. The structure itself requires each loop level described via the IterBasis structure, as shown in lines 14–17. The array and loop parameters
assumed and set by the structure are identical to those listed in Table 1 and 2 except that the array element size is 16 bytes instead of 8 and the step size of level 1 is 2 to avoid padding each element for alignment purposes. Once each loop level is described, the IterSpace structure is initialized with the size of the array element using init_iterspace() in line 18. Strides are pre-calculated during initialization and used later by the block access list generator together with Iterator.

An iteration space is decomposed and distributed to SPEs as evenly as possible after being initialized. Lines 19–21 perform iteration space decomposition. Iterator is used for each access pattern to efficiently iterate over the iteration space. In lines 23–26, iterators (itl/ito) are allocated with pre-initialized parameters (is) and set to the iteration range assigned to the SPE by compute_bounds(). In this example, the iteration space is same for all access patterns and is is shared.

The MonoChromatic mci is allocated for two in arrays, A and B, in line 28 and used for efficient handling of the arrays accessed in the same pattern. mco is allocated for the out array C in line 31.

The set_wave() interface is used to set parameters for individual arrays in a monochromatic group with the effective address (X.info.ea), the offset (0), the block descriptor (sblk) and the amplitude (1) in line 29, 30 and 32. The kernel written by the programmer is plugged in the framework in line 34. It is repeatedly called by the buffer manager whenever a local buffer is ready with new data. The kernel requires one user parameter dt. It is passed from PPE to SPE via the signaling framework of Strider and pushed into the user parameter stack in line 35, so that the kernel can pop and use the parameter. Finally, in line 36, the buffer manager strider_nlm_f_pn() is called for the monochromatics associated with the in arrays and the out array. The buffer manager type expects a fundamental monochromatic in a perfectly nested loop and is optimized for this purpose. The fundamentals defined separately for in, out and inout are passed to the buffer manager via three parameters. However, in this example the inout parameter is not used. The transfer of inout arrays is scheduled inside the buffer manager using double-buffering. The last part of the code performs deallocation of structures.

The kernel kernel_nested_loop() shown in Figure 5 follows the user kernel prototype of Strider. The parameter list includes the length and address of the buffer and the first iteration that accesses data in the buffer. The kernel is composed of two parts, parameter parsing and core computation. The former handles the parameters passed in the kernel parameter stack. The latter performs the actual computation on the buffered data. Some of the parameters do not change during a run of the buffer manager. To avoid parsing redundant parameter values, such parameters are handled only once when the active buffer manager calls the kernel for the first time. In the example shown in Figure 5, one user parameter dt is parsed once in line 8.

The problem with the code shown in Figure 3 is that the transfer block is only two array elements of type double. Thus, there is a large number of transfers, carrying significant overhead. An alternative is shown in Figure 4. In this approach, the array is considered as a virtual two-dimensional array with a larger element size as shown in Table 3, and the original 4-level loop is converted to a 2-level loop as shown in Table 4. The transfer block size becomes $(nz - 2) \times 6 \times 8$ bytes which is set in line 13 in Figure 4. This size is actually smaller than the element size of the virtual array. Thus, the offset is used to adjust the starting address of the transfer block in lines 28, 29, and 31 in Figure 4, to facilitate folding the original 4-dimensional loop into 2 dimensions.

### Table 3: Array description for the example in Figure 4

<table>
<thead>
<tr>
<th>number of dimensions</th>
<th>dim 2 size</th>
<th>dim 1 size</th>
<th>element size</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>nx</td>
<td>ny</td>
<td>nz $\times$ b $\times$ h</td>
</tr>
</tbody>
</table>

### Table 4: Loop parameters for the example in Figure 4

<table>
<thead>
<tr>
<th>induction variable</th>
<th>array dimension</th>
<th>start</th>
<th>end</th>
<th>step size</th>
<th>loop level</th>
<th>step direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>2</td>
<td>0</td>
<td>nx</td>
<td>ny-1</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

5. Experimental analysis

5.1. Experimental platform

Our experimental platform is an IBM QS20 blade. The blade runs Linux Fedora Core 7 with kernel version 2.6.22-5 for the 64-bit PowerPC architecture. The blade has two Cell processors and 1 GB of off-chip main memory. The reported timing breakdowns are obtained using the time base register on PPE. We used the IBM Cell SDK 3.0, ppu-gcc 4.1.1, and spu-gcc 4.1.1 [7].

---

void kernel_nested_loop(uint32_t buf,
uint32_t buflen, uint32_t *idx)
{
uint32_t i;
static double dt;

if (__builtin_expect(get_param_at(I_RESET),0)) {
  dt = pop_double_param();
  set_param_at(I_RESET, 0);
}

volatile double *A, *B, *C;
A = ((double**)((uint32_t*)buf)[IBUFIDX])[0];
B = ((double**)((uint32_t*)buf)[IBUFIDX])[1];
C = ((double**)((uint32_t*)buf)[IBUFIDX])[0];
for (i=0; i<buflen ; i++)
C[i] = dt*A[i] + B[i];

---

Figure 5: The user kernel for the loop shown in Figure 2
5.2. Experimental results

We experimented with two kernels which transpose and copy 2-dimensional arrays of doubles respectively. We evaluate the kernels with three different array sizes, namely $600 \times 600$, $1200 \times 1200$, and $1800 \times 1800$, using up to 16 SPEs. Each kernel is repeated 1,000 times and each reported data point is the average of 10 runs. We used several block shapes of the same total size (1,200 array elements): $600 \times 2$, $300 \times 4$, $100 \times 12$, $40 \times 30$, $30 \times 40$, $12 \times 100$, $4 \times 300$, and $2 \times 600$. Two decomposition strategies were used as discussed in Section 3.2. The first strategy decomposes the entire range of iterations across all loop levels, while the second strategy decomposes only the iterations of the outermost loop. For a given block shape of size $a \times b$, the larger the $a$, the longer the DMA list and the longer the time it takes to prepare and handle the list. Therefore, “taller” and “thinner” blocks are more likely to take longer to transfer than “shorter” and “thicker” blocks. Figure 6(a) and Figure 6(b) verify this claim. In row copy, the outer loop iterates over the column direction. Thus, when the $600 \times 2$ block is used for a $600 \times 600$ array and only the outer loop is parallelized, a single SPE copies the entire array which is split in 300 blocks and the workload is highly unbalanced. On the other hand, when iterations across all loop levels are decomposed, each of 4 SPEs takes 18 blocks and each of the other 12 SPEs takes 19 blocks. The difference is shown with the the data points of $600$-outer and $600$-all for block size $600 \times 2$ in Figure 6(a). In column copy, the most unbalanced case occurs for block size $2 \times 600$, as shown in Figure 6(b).

To provide a baseline for comparing the performance of Strider, we present the performance of the copy and transpose kernels implemented with Sequoia. In Se-
quoia, the parallelization of one or more loop levels dictates the block shape and the data decomposition method and is defined statically at compile time. In Strider, such a decision is made at runtime and the parallelization method can be recalibrated on the fly. To achieve the maximum degree of symmetry between the Sequoia and Strider implementations, each loop level in the Sequoia implementation is split 4 ways, therefore the 2-and Strider implementations, each loop level in the Sequoia implementation is split 4 ways, therefore the 2-level loops are split 16 (4 × 4) ways. Sequoia’s additional copy overhead on PPE, discussed in Section 3.1.2, is not accounted for in the results. Strider takes on average 71% of the time taken by Sequoia to copy and transpose arrays, as shown in Figure 6. The overall bandwidth utilization of Strider is 1.76 times that of Sequoia.

Figure 7 compares the performance of copying columns and transposing columns to rows between Strider and Sequoia. Parameters in these experiments were selected favorably for Sequoia. For example, the chosen block size (30 × 40) works best for Sequoia but not always for Strider. The Sequoia implementation is also recompiled for each data point with the best loop parallelization strategy. Strider performs better than Sequoia because Sequoia does not divide the entire iteration space across multiple levels of a nested loop. Sequoia only parallelizes within each level, while the code generated from the Sequoia compiler can not adapt to a different block shape at runtime. In addition, Strider performs better even if only the outer loop is decomposed, for block shapes that are neither too thin, nor too thick. The difference stems from the additional overhead in Sequoia’s generated kernel code for addressing each data element in local storage, through Sequoia’s array representation structure.

In the transpose kernel, the shape of blocks read is different from that of blocks written. The decomposition of the iteration space is based on the input array and the shape of the input block. Figure 6(c) shows the load imbalance observed also in the case of copy (Figure 6(a)). Figure 6(d) shows similar performance trends as Figure 6(b). In Figure 6(c) and 6(d), the performance degradation due to longer DMA list of input block and output block is shown with the leftmost data points and the rightmost data points respectively.

Figure 8 shows the superior bandwidth scaling of Strider when multiple input arrays of 2048 × 2048 integers are added and transposed into an out array from row major to column major using blocks of size 32 × 32 and 16 SPEs. This result shows the effectiveness of Strider’s access pattern grouping.

6. Future work

Planned future work involves supporting block tiling and handling of loops with loop-carried dependencies in Strider. In particular we plan to extend Strider to support stencil computations [8], using two approaches, namely pipelining and wavefront execution [9]. Pipelining can be supported with only slight modifications for communication and synchronization between SPEs and between SPEs and the PPE. Reducing or eliminating the overhead of transferring redundant data requires further investigation in this context. Wavefront execution requires support for more general affine loop bounds and can be achieved as discussed in Section 3.1.3. Another direction of improvement is to evolve the current library framework into a code generation framework utilizing polyhedral loop analyzers [10].

7. Conclusions

Strider is a library framework for programming applications that access multi-dimensional arrays on multi-core processors with explicitly managed memories. Strider offers higher abstraction than existing library frameworks and higher flexibility than existing code generation frameworks, while also sustaining higher performance.

Acknowledgments: This work has been supported by grants from NSF (CCF-0715051, CNS-0720750, CNS-0720673) and the European Commission (FP7-PEOPLE-224759, FP6-FET-27648, FP6-IST-217068, FP6-IST-004408). We thank the Barcelona Supercomputing Center (BSC) for providing us access to Cell blades.

References