Automatically Optimizing Stencil Computations on Many-core NUMA Architectures

Pei-Hung Lin[1] Qing Yi[2] Daniel Quinlan[1] Chunhua Liao[1] Yongqing Yan[2]

[1] Lawrence Livermore National Laboratory, Livermore CA 94550, USA
[2] University of Colorado, Colorado Springs, CO 80918, USA

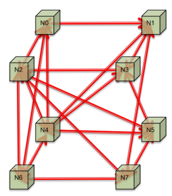
Abstract. This paper presents a system for automatically supporting the optimization of stencil kernels on emerging Non-Uniform Memory Access(NUMA) many-core architectures, through a combined compiler + runtime approach. In particular, we use a pragma-driven compiler to recognize the special structures and optimization needs of stencil computations and thereby to automatically generate low-level code that efficiently utilize the data placement and management support of a C++ runtime on top of NUMA API, a programming interface to the NUMA policy supported by the Linux kernel. Our results show that through automated specialization of code generation, this approach provides a combined benefit of performance, portability, and productivity for developers.

1 Introduction

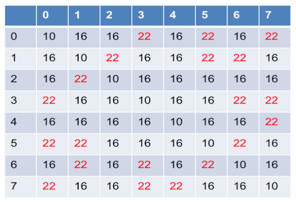
Modern architectures increasingly use a large number of cores to boost application performance. To reduce the cost of using a global bus to support cache coherence, these cores are typically decomposed into a hierarchy of NUMA nodes, illustrated in Fig. 1(a). Fig. 1(b) shows the widely varying memory latencies across the eight NUMA cores in a single compute node. To attain high performance, applications need to be aware of these different latencies to reduce the overhead of remote data accesses. In addition to the obvious performance benefit, such a design offers potential portability to future architectures, which may no longer support cache coherence across different NUMA nodes.

It is well known that significant developer effort is required to decompose an application into separate memory spaces and then explicitly reference remote data based on their locations. Instead of burdening developers with the effort, which degrades their productivity, we propose an automated approach, where user applications are written using a conventional SMP programming model, e.g., OpenMP [18], and a compiler is used to automatically translate the high level specifications down to a lower level implementation that explicitly manages local and remote memory references and by invoking a runtime library, flexibly manages the distribution and relocation of data.

This paper presents such a compiler and runtime combination for an important class of scientific kernels, the stencil computations, which are generally considered one of the most fundamental kernels of scientific simulations and are



(a)



	0	1	2	3	4	5	6	7
0	10	16	16	22	16	22	16	22
1	16	10	22	16	16	22	22	16
2	16	22	10	16	16	16	16	16
3	22	16	16	10	16	16	22	22
4	16	16	16	16	10	16	16	22
5	22	22	16	16	16	10	22	16
6	16	22	16	22	16	22	10	16
7	22	16	16	22	22	16	16	10

(b)

Fig. 1. AMD 6380 CPU: (a) NUMA hierarchy; (b) latency distance matrix

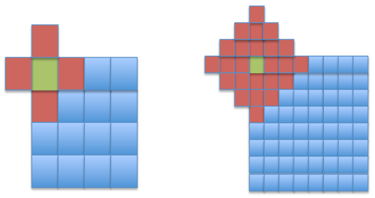


Fig. 2. Stencil distance in a 2D representation

widely used in solving problems such as partial differential equations. As illustrated by Fig. 2, a typical stencil kernel iteratively modifies each element of a regular grid based on values of its neighboring elements. The number of neighbors in the computation can vary significantly depending on the distances of the neighbors and the dimension of the grid.

The regular structures of the stencils make them ideal candidates for high performance computing on the latest ccNUMA (cache coherent NUMA) architectures, which use inter-node communication between cache controllers to support cache coherence across cores. A principle to obtaining high performance on such systems is to have each core perform local computations most of the time and to limit cache coherence induced traffic among neighboring nodes, thereby avoiding traffic congestion. Such constraints are naturally satisfied by decomposing the underlying grid of a stencil kernel, e.g., the one in Fig. 2, into blocks so that the composition of the grid matches the underlying topology of the NUMA cores. Then, each block of data, together with the computation that modifies it, can be permanently allocated to its corresponding NUMA core, eliminating global data movements and maintaining a consistent level of locality throughout.

We have developed a pragma-driven compiler and a runtime library to accomplish the above for stencil computations. The compiler is designed to recognize the special structures of stencil computations and thereby to automatically generate low-level C code that explicitly distinguishes local and remove memory references to efficiently utilize the underlying data placement and management scheme supported by a runtime library on top of NUMA APIs (libnuma) [12]. We show that through automated specialized code generation for stencils, our approach provides a combination of good performance, architecture portability, and productivity for developers. Our technical contributions include:

– We present how to use specialized code generation to take advantage of the structures of stencil codes and thereby automatically provide advanced optimization support for these kernels on NUMA architectures.
– We study the implications of varying optimization schemes for ccNUMA and demonstrate the importance of coordinated compiler and runtime support.

The rest of the paper is organized as follows. Section 2 presents the programming interface of our system. Sections 3 and 4 present our stencil compiler and runtime library. Section 5 presents experimental results. Section 6 discusses related work. Section 7 summarize our conclusions.

```
1: #pragma stencil s1 time <t> array [X*Y] <A0,Anext>
2: for (t = 0; t < timesteps; t++) {
3:   if (t%2 == 0) { old_a = A0; a = Anext; }
4:   else {a = A0; old_a = Anext; }
5: #pragma stencil s1 data <i,j> array <old_a,a>  halo <-1,1> <-1,1> copy_halo
6:   for (int i = 1; i < X-1; i++)
7:     for (int j = 1; j < Y-1; j++)
8:        old_a[i][j] = (a[i-1][j]+a[i+1][j]) + a[i][j-1] + a[i][j+1]))/4.0;
9: }
```

Fig. 3. Example: 2-D stencil with the optimization pragma

2 The Programming Interface

The programming interface of our system includes a set of pragma notations, illustrated by lines 1 and 5 of Fig. 3, to describe various properties of a stencil. Each pragma identifies an immediately following loop as part of a stencil computation to be optimized, by specifying the following properties of the stencil.

- A name that uniquely identifies the stencil kernel, so that multiple pragmas can be used to collectively define a single stencil. For example, both pragmas in Fig. 3 use $s1$ as the stencil name, with line 1 specifying the time dimension of the stencil, and line 5 the data dimensions.
- The time dimensions of the stencil, expressed using the notation $t$ime<v_0 ...v_m>, where each $v_i(i = 0, ..., m)$ specifies the index variable of a loop that enumerates a time dimension of the stencil. In Fig. 3, the time dimension is enumerated by the t loop at line 2 and specified inside the pragma at line 1.
- The data dimensions of the stencil, expressed using the notation $d$ata <v_0 ... v_n>, where each $v_i(i = 0, ..., n)$ specifies the index variable of a loop that enumerates different elements of a stencil data dimension. In Fig. 3, the data dimensions are enumerated by the i and j loops at lines 6-7 and are specified inside the pragma at line 5.
- The names and dimension sizes of the arrays used to store the stencil data, specified using the notation $a$rray [d_1*...*d_m] <a_0,...,a_l>, where $d_1 * ... * d_m$ defines the dimensionality (m) of the stencil, the size of each dimension, and the name $a_i(i = 0, ..., l)$ of each array, as illustrated by the [X*Y] <A0,Anext> and <old_a,a> declarations at line 1 and line 5 of Fig. 3.
- The neighboring references used to update each element of the stencil, specified using the notation $h$alo <l_0,r_0>...<l_n,r_n>, where each <l_i,r_i> (i=1,...,n) specifies the neighbors from the left (l_i) and right (r_i) of data dimension i. For example, the simple stencil in Fig. 3 uses four neighbors, one from each side of each dimension, to update each element.
- Optimization configurations, e.g., *halo_copy* or *halo_no_copy* (the default option) to indicate whether to pre-copy values of the neighboring references to local variables before using them to update halo regions of each core.

Each pragma defines the immediately following loops as new components of the stencil. Pragmas with the same identifier, which can span multiple procedures, are required to collectively satisfy the following constraints.

- If a stencil has multiple pragmas, the data dimensions must be nested inside the time dimensions. In particular, when across procedures, each inner

pragma must be inside a procedure invoked by the body of an outer one, and the function that contains the inner pragma must not be invoked elsewhere for other purposes (a function specialization pre-transformation can be applied by the compiler to automatically support this property).
- Each pragma may introduce a set of arrays used to store the stencil data. All stencil arrays must have the same size, and when multiple sets of arrays are introduced, each inner declaration introduces a set of new names aliased to those introduced by the outer pragmas.
- When modifying each element at subscript $(v_0, ..., v_n)$ of a stencil array, the computation only uses elements from the other arrays that are within the neighborhood of $(v_0 + l_1, ..., v_n + l_n)$ and $(v_0 + r_1, ..., v_n + r_n)$, where (l_i, r_i) i=0,...,n is the hallo region of each dimension.

The compiler relies on the above properties to ensure safety and profitability of optimizations. In particular, if the stencil modifies an array a_i by reading only neighbors from the other arrays, no synchronization is needed when using OpenMP to parallelize the data dimension loops within each time step. On the other hand, if a_i is modified by reading its own neighborhood, additional synchronizations are needed to make sure up-to-date values of a_i are used. Our system currently support only the first case (a.k.a. the jacobi type of stencils).

3 The Stencil Compiler

Fig. 4 shows the algorithm implemented by our compiler to automatically convert an annotated stencil kernel into its lower level implementation (llustrated in Fig. 5) through the following three steps inside function *transform-stencil*.

- Data placement: decompose and copy the data onto the proper NUMA cores, illustrated by lines 2-7 of Fig. 5, which create two new arrays of type MulticoreArray<float>, a distributed array type defined in our runtime library with pre-allocated data on the NUMA cores, and then concurrently copy the original data into these distributed arrays. The original stencil code is then modified to use the new arrays, named *_Anext* and *_A0* In Fig. 5.
- Many-core computing: deploy the NUMA cores, which have been pre-allocated with stencil data in the constructor of the MulticoreArrays at line 2 of Fig. 5, to each concurrently update their pre-allocated data (lines 14-18 of Fig. 5).
- Data collection: copy the distributed data at each NUMA core back to the original stencil arrays, illustrated at lines 21-24 of Fig. 5.

The data placement and collection steps serve the purposes of copying data back and forth between the original and the distributed stencil arrays. They represent the most significant overhead of the parallelization optimization and are placed outside of the outermost time loop of the annotated stencil computation, so that the overhead can be amortized when the computation is repeated many times (which is typical in practice). As summarized at lines 16-17 of Fig. 5, the many-core computing step extracts the following two components of the computation to be performed on each NUMA core.

```
transform-stencil ($p$ : stencil pragma, input: stencil code to modify)
1:    $t$=time_loop($p$); $arr$=stencil_arrays($p$,$t$); insert gen-distribution(p,arr) before t in input;
      **for each** array $a \in arr$ **do**: replace $a$ with $multicore\_arr(a)$ in $t$; **enddo**;
      **for each** function g called inside $t$ and parameter $x$ of $g$ s.t. $x \in stencil\_arrays(p)$ **do**
        replace $x$ with multicore_array(x) in $g$; **enddo**
2:    $d$ = data_dims($p$); $local$=gen-stencil-local($p$, $d$); $edges$=gen-stencil-boundaries(p,local,d);
      replace $outermost\_loop(d)$ with gen_manycore_compute(local, edges) in input;
      **if** configured to do so **then** apply_aray_copying_opt(input, stencil_arrays(p,d), hallo(p)) **endif**
3: insert gen-data-collection(p, arr) after t in input;

gen-stencil-local($p$: stencil pragma, $d$: stencil data dimensions)
2.1: res = copy(outermost_loop(d));
2.2: **for each** loop $l \in d$ **do**:
       $local_l$ = replace stencil_size($l$) with local-size($p$, $l$) in $l$; replace $l$ with $local_l$ in $res$; **enddo**
2.3: **for each** $a[sub] \in res$ s.t. $a \in stencil\_arrays(p)$ **do**: replace a[sub] with local_ref(a[sub]); **enddo**
     return $res$;

gen-stencil-boundaries($p$: stencil pragma, $local$: local computation, $d$: stencil data dimensions)
    res=empty; cdims = $\emptyset$;
    **for each** loop $l$ : for $i = lo_i..hi_i \in d$ s.t. $halo(p,l) = (halo_l, halo_r)$ **do**
2.4:  $left_l = right_l$ = empty;
2.5:  **for each** iteration $v$ of $l$ s.t. $lo_i + halo_l \le v \le lo_i$ **do**
        $left_v$ = replace $i$ with $v$ in copy(body_of $l$); append $left_v$ to the end of $left_l$; **enddo**
      **for each** iteration $v$ of $l$ s.t. $hi_i \le v \le hi_i + halo_r$ **do**
        $right_v$ = replace $i$ with $v$ in copy(body of $l$); append $right_v$ to the end of $right_l$; **enddo**
2.6:  left_edge=replace $l$ with $left_l$ in copy(local); right_edge = replace $l$ with $right_l$ in $copy(local)$;
2.7:  **for each** $a[sub]$ in $left\_edge$ s.t. $a \in stencil\_arrays(p)$ and offset(sub,l) $< lo_i$ **do**
        replace $a[sub]$ with $remote\_stencil\_from\_left(r)$; **enddo**
      **for each** $a[sub]$ in $right\_edge$ s.t. $a \in stencil\_arrays(p)$ and offset(sub,l) $> hi_l$ **do**
        replace $a[sub]$ with $remote\_stencil\_from\_right(r)$; **enddo**
2,8:  append gen-stencil-boundaries($p$,$left\_edge$,$cdims$) with conditional at the end of res;
      append gen-stencil-boundaries($p$,$right\_edge$,$cdims$) with conditional at the end of res;
      $cdims = cdims \cup \{l\}$;
    **enddo**
    return res;
```

Fig. 4. Algorithm: transforming stencil computations

The inner stencil, which modifies and reads only data that are on the local core, illustrated by lines 3-5 of Fig. 6. The inner stencil is extracted by invoking the *gen-stencil-local* algorithm in Fig. 4, which includes three steps: (2.1) make a copy of the outermost data dimension loop, e.g., loop i in Fig. 3, which represents a single time step iteration of the stencil; (2.2) modify the copy so that the upper bound of each data dimension (e.g., X and Y in Fig. 3) is replaced with the size of the local portion of the dimension (e.g., $c.sz0$ and $c.sz1$ in Fig. 6); and (2.3) replace references to the global stencil arrays (e.g., _old_a and a in Fig. 5) to instead use their local copies (e.g., _old_a_local and _a_local in Fig. 6).

The stencil boundaries, which modify data located at either end of a data dimension and need to read data from the neighboring cores to correctly perform the updates. These boundary computations are extracted by invoking the *gen-stencil-boundaries* algorithm in Fig. 4. Since two boundaries at both ends must be considered for each data dimension, 2^n cases are generated for a stencil of n dimensions. As example, the boundary cases of the two-dimensional stencil in Fig. 3 are illustrated at lines 6-23 of Fig. 6. The algorithm in Fig. 4 uses a variable *cdims* to keep track of all the data dimensions already processed and uses two variables, $left_l$ and $right_l$, to store computations that modify either end (left or

```
/* initialize local arrays and their dimensions on each core*/
MulticoreArray<float> _Anext(nz,ny,nx,CORE_NUM,0,true), _A0(nz,ny,nx,CORE_NUM,0,true);
int numberOfCores = _A0.get_numberOfCores();
#pragma omp parallel for private(i,j,k)
for (int core = 0; core < numberOfCores; core++) {
      ... copy Anext and A0 into _Anext and _A0 ...
  }
MulticoreArray<float>* _old_a, *_a;
#pragma stencil s1 time <t>  array <_A0,_Anext>
for (t = 0; t < timesteps; t++) {
  if (t%2 == 0) { _old_a = _A0; _a = _Anext; }
  else {_a = _A0; _old_a = _Anext; }
  #pragma stencil s1 data<i,j> array<_old_a,_a>  halo<-1,1><-1,1> dist<blocked>
  #pragma omp parallel for private(i,j,k)
  for (int core = 0; core < numberOfCores; core++) {
  ...compute 2D stencil on the local arrays of each core ...
  ...compute boundary values by communicating with the neighbors ...
  }
}
 #pragma omp parallel for private(i,j,k)
 for (int core = 0; core < numberOfCores; core++) {
       ... copy _Anext and _A0 back into Anext and A0 ...
  }
```

Fig. 5. Example: structure of lower-level implementation of Fig. 3

right) of each data dimension l (step 2.4). Each boundary case is extracted from loop l by removing the loop and replacing its index variable with an iteration number in the left or right halo region of the dimension (step 2.5). Next, the original l loop in the inner stencil is replaced with a corresponding boundary case to generate $left\_edge$ and *right_edge*, which contain unrolled halo iterations of l on the left and right boundaries respectively (step 2.6). Then, step 2.7 replaces the stencil array references that are outside the local core with remote references that explicitly fetch the data from the neighboring cores. Finally, for each stencil boundary computation already generated and saved in $left\_edge$ and *right_edge*, invoke the *gen-manycore-stencil-boundary* function again to generate computations at the corners of multiple distributed data dimensions (step 2.8), which need to access remote data from two or more neighbors. All boundary cases are then wrapped inside a sequence of if conditionals, shown at lines 6, 9, 12, 15, 17, 19, 22, and 24 in Fig. 6, before being appended to the result.

The low-level implementation in Fig. 6 is essentially the result of numerous splitting and unrolling transformations to the nested data dimension loops of the stencil. Each split loop nest contains a unique combination of local and remote data references, with each remote reference triggering a data movement between a pair of neighboring cores. Two benefits are offered by such an implementation. First, the implementation knows and explicitly enumerates the exact location of each data item and thus incurs no runtime address translation overhead and requires no cache coherency support from the hardware. Second, the separation of different combinations of local vs. remote references allows additional optimization opportunities, e.g., by prefetching the remote references explicitly, shown as the last operation of step (2) of the algorithm in Fig. 4. The complexity of the low-level implementation, while nearly impossible for a developer to manually manage, is easily managed by compilers by recursively enumerating all the boundary cases, as demonstrated, enhancing application portability.

```
float *_a_local = (*_a).arr_ptrs[core], *_old_a_local = (*_old_a).arr_ptrs[core];
Core<float>& c = (*_a).core_info[core];
for (i=1; i<c.sz0-1; i=i+1) /* computation with only local references */
  for (j=1; j<c.sz1-1; j=j+1)
    _old_a_local[i+j*c.sz0] = (_a_local[i-1+j*c.sz0] + _a_local[i+1+j*c.sz0] +
                             _a_local[i+(j-1)*c.sz0] +_a_local[i+(j+1)*c.sz0])/4.0;
if (!c.is_leftmost_core[0]) /* left boundary computation at dimension 0*/
    for (j=1; j<c.sz1-1; j=j+1)
      _old_a_local[j*c.sz0] = ((*_a).arr_ptrs[c.l_neighbor[0]] [(c.l_sz0-1)+j*c.l_sz0]
      +_a_local[(1+j*c.sz0]+_a_local[(j-1)*c.sz0] +_a_local[(j+1)*c.sz0])/4.0;
if (!c.is_rightmost_core[0]) /* right boundary computation at dimension 0*/
    for (j=1; j<c.sz1-1; j=j+1)
      _old_a_local[c.sz0-1+j*c.sz0] = (_a_local[c.sz0-2+j*c.sz0]+(*_a).arr_ptrs[c.r_neighbor[0]]
      [1+j*c.r_sz0] +_a_local[c.sz0-1+(j-1)*c.sz0] +_a_local[c.sz0-1+(j+1)*c.sz0])/4.0;
if (!c.is_leftmost_core[1]) /* left boundary computation at dimension 1*/
    for (i=1; j<c.sz0-1; i=i+1)
      _old_a_local[i] = _a_local[i-1] +_a_local[i+1]
      +(*_a).arr_ptrs[c.l_neighbor[1]] [i+(c.l_sz0-1)*c.l_sz0]+_a_local[i+c.sz0)/4.0;
    if (!c.is_leftmost_core[0]) /* if core is additionally on the boundary at dimension 0 */
      { ...... left-left corner computation ... }
    if (!c.is_rightmost_core[0]) /* if core is additionally on the boundary at dimension 0 */
      { ...... right-left corner computation ... }
if (!c.is_rightmost_core[1]) /* right boundary computation at dimension 1*/
    for (i=1; i<c.sz0-1; i=i+1)
      _old_a_local[i+(c.sz1-1)*c.sz0] = (_a_local[i-1+(c.sz1-1)*c.sz0]+_a_local[i+1+(c.sz1-1)*c.sz0]
      +_a_local[i+(c.sz1-2)*c.sz0] +(*_a).arr_ptrs[c.r_neighbor[1]][i])/4.0;
    if (!c.is_leftmost_core[0]) /* if core is additionally on the left boundary at dimension 0 */
      { ...... left-right corner computation ... }
    if (!c.is_rightmost_core[0]) /* if core is additionally on the right oundary at dimension 0 */
      { ...... right-right corner computation ... }
```

Fig. 6. Example: local and boundary computation per core

4 Runtime Support

Our runtime library provides a C++ abstraction, the *MulticoreArray* templated class used at line 2 of Fig. 5, to support NUMA-aware stencil computation. The abstraction internally integrates the thread decomposition and scheduling support in OpenMP with data placement support through libnuma. Each OpenMP thread is bound to a hardware core, through the system library *sched_setaffinity*(). The NUMA topology is referenced by the runtime for the binding of OpenMP thread and hardware core, and libnuma is invoked in the constructor of the abstraction to allocate a distributed stencil array of the desired data dimensions, with its internal data placed onto a pre-specified number of different hardware cores. To minimize remote memory access latency, neighboring stencil data are allocated either on hardware cores located inside the same NUMA node, or cores that belong to adjacent NUMA nodes. Halo region copying is supported to help developers manage data movement across cores.

4.1 Thread Decomposition And Management Using OpenMP

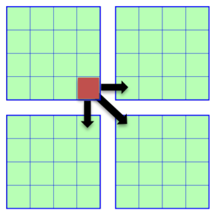
Our runtime allows the number of hardware cores to be used for each data dimension of the stencil to be specified when invoking the constructor of the MulticoreArray abstraction. If unspecified, the maximal number of cores that match the underlying system topology is used. and each core is allocated with blocks of distributed data to be used for later computation. The runtime relies on libnuma to retrieve NUMA distances, a relative distance in the machine topology between two NUMA nodes, among all available NUMA nodes. A multi-dimensional topology can be constructed with the available NUMA information.

Based on the core numbers to be used for each stencil dimension, the runtime use the *omp_set_num_threads()* and *omp parallel for* clause to setup the parallelization environment. The system call *sched_setaffinity()* is used to enforce CPU affinity and bind the CPU core to a designated OpenMP thread. For example, given a fully parallelized configuration using 64 OpenMP threads on a 64-core machine, OpenMP threads with ID 0 to 7 will be bound to hardware cores with ID 0 to 7. These 8 hardware cores reside in NUMA node 0 according to the NUMA information from the hardware specification. This thread binding is different from the default OpenMP support, which binds an OpenMP thread to any available hardware core based on the system status. In contrast, our runtime exerts full control in the thread scheduling for the many-core hardware.

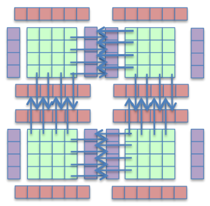
4.2 NUMA-aware Data Placement

Our runtime uses internal data structures inside the *MulticoreArray* abstraction to decompose a stencil array into a collection of sub-arrays. Each sub-array separately stores the stencil data to be operated in a designated thread and is stored in a continuous memory space on a hardware core, together with additional information about the size of the local data and pointers to data that belong to its neighboring threads. Multi-dimensional distribution is used to distribute blocked data to the sub-arrays. By default, each sub-array (except the last one) contains the same number of distributed elements. The runtime then evenly assigns sub-arrays to OpenMP threads based on the sub-array IDs and OpenMP thread IDs. Sub-arrays with adjacent ID numbers are assigned to the same NUMA node or adjacent NUMA nodes when possible. After the assignment, the function *numa_alloc_local()* from libnuma is called by each thread to allocate memory space for the distributed data, thereby enforcing all the distributed data are allocated to their designated OpenMP threads and hardware cores/NUMA nodes. When the number of sub-arrays is more than the available hardware cores, our runtime assigns multiple sub-arrays with neighboring IDs to the same OpenMP thread. This again enforces that adjacent sub-arrays are allocated to the same or neighboring NUMA nodes to reduce memory references crossing NUMA nodes. All the threads use the *numa_alloc_** functions from libnuma to allocate local memory for their data.

Data elements inside a *MulticoreArray* object can be accessed in two different ways: (1) through a high-level interface that allows data to be accessed based on their locations in the original stencil arrays using subscript notations, with the subscripting operator internally translating the global coordinates to the appropriate sub-arrays and local subscripts within the subarrays; and (2) through the low-level interface, which directly references the sub-array pointers and their local elements and is therefore much more efficient. The high-level interface is provided to the developers for convenience, while the low-level interface is used by our compiler, illustrated in Fig. 6, to ensure efficiency of the generated code,



(a) No local halo



(b) Detached halo

Fig. 7. Halo management

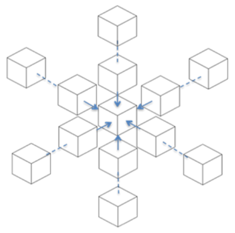


Fig. 8. 3D Stencil

Our runtime is specialized for the NUMA architecture and is different from the default OpenMP runtime in two key aspects: (1) it decomposes data based on the NUMA topology to minimize exchanges across NUMA nodes, whereas OpenMP uses the first-touch policy; and (2), it supports multi-dimensional data decomposition, with data elements in each decomposed sub-array residing in adjacent memory spaces for better spacial locality. Through the pragma-driven programming interface, our combined runtime and compiler support allows developers to inject domain-specific knowledge into the data and computation decomposition process to maximize application performance.

4.3 Halo data management

As stencil data are distributed onto different cores and updated concurrently, each thread needs data from its neighbors to update elements on the boundaries of its local block. These neighboring data are called halo regions of each thread. Our system supports two approaches to managing halo regions, illustrated in Fig. 7. The first approach (shown in Fig. 7(a)) keeps the halo regions in the remote memory, and the second (Fig. 7(b)) replicates the remote data on the local core. The different storage forms impact the performance of the computation by changing when the remote data is fetched (e.g., just in time before they are used vs. far ahead of time using detached halo). Further, hardware with small cache capacity may prefer no local halo storage to save space. The first halo management approach, with computation fetching halo data remotely from the neighboring cores, is adopted by the compiler generated code shown in Fig. 6.

5 Experimental Results

We implemented our stencil compiler by combining the POET program transformation language [24] with the ROSE C/C++ compiler [19]. The compiler is evaluated by using it to automatically generate low-level implementations for four 3D stencil kernels, with 7-point, 13-point, 19-point, and 25-point updates respectively. In particular, each kernel repetitively modifies two 3D stencil arrays of the same size, with each element of one array modified using neighboring elements of the other. The 7-point stencil updates each element using two neighboring elements from each dimension of the other array, as illustrated by the 2D stencil code in Fig. 3. The 13-point stencil updates each element using four neighboring elements of each dimension, as illustrated in Fig. 8, and so on. Each kernel has a baseline OpenMP implementation, where a single OpenMP parallel for pragma is used to parallelize the outermost data dimension loop of each stencil. In contrast, the implementation generated by our compiler (the stencilOpt version) parallelizes all the data dimensions of the stencil instead of just the outermost one as the preferred configuration. Further, each stencil array is placed explicitly on the appropriate NUMA cores, and system-level affinity binding is used to ensure each thread only modifies its local data. The baseline OpenMP implementations have their data distributed among NUMA nodes following the first-touch policy. The OpenMP thread affinity is also setup in OpenMP environment. In contrast, the stencilOpt implementations use the data distribution strategy described in Section 4.2.

All kernel implementations are evaluated on a 64-core AMD 6380 workstation comprised of four sockets (16 cores per socket). Each core has a 16 KB L1 data cache, and every two cores share a 2 MB L2 cache memory. Every 8 cores form a NUMA node and share a 8 MB L3 cache. The NUMA distance matrix in Fig. 1 shows the relative memory latencies among the different NUMA nodes, and Fig. 1(a) shows the structure of the NUMA hierarchy. All implementations were compiled on the machine using gcc with -O2 option. Each implementation is evaluated five times, and its average performance is reported. The performance variations across different runs are generally under 3%.

5.1 The Overall Performance

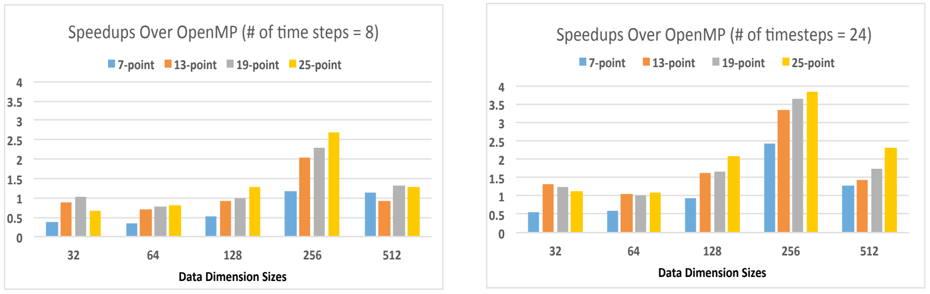


Fig. 9. Speedups attained by our compiler over baseline OpenMP implementations

Fig. 9 shows the speedups attained by using our compiler to automatically generate low-level implementations (the *stencilOpt* implementation) for the four 3D stencil kernels, when the size of each data dimension ranges from 32 to 512 and the number of time iterations from 8 to 24. From these results, when the number of time iterations is 24, our *stencilOpt* implementation were able to perform better than the OpenMP implementation in all cases except for the 7-point stencil, which has the fewest neighboring data references, where the stencilOpt implementation performed worse than the OpenMP one when the problem sizes are ≤ 128. When the number of neighboring references increases, the stencilOpt implementation has uniformly attained a speedup, ranging from factors of 1.1 to 3.7, over the OpenMP implementation. In most cases when the problem size or the number of neighboring references increases, so does the performance speedup over the OpenMP implementation. The overall results indicate that the stencilOpt implementation can manage memory and neighboring core communications much better than the baseline OpenMP implementation, indicating the effectiveness of NUMA-aware data placement by our runtime.

When the number of time iterations equals 8, the speedups attained by stencilOpt generally follow a similar pattern but are much worse than the $time = 24$ cases. This is because when many fewer iterations of the stencil computation are repeated, there are insufficient reuses of the distributed stencil arrays to compensate for the extra overhead of constructing the distributed stencil arrays and copying back and forth between the original stencil arrays and the distributed ones. When amortized at least 24 times, which are much smaller than the number of time iterations in realistic applications, the overhead is no longer a significant factor in performance, and significant speedups can be attained.

5.2 Impact of Execution Configurations

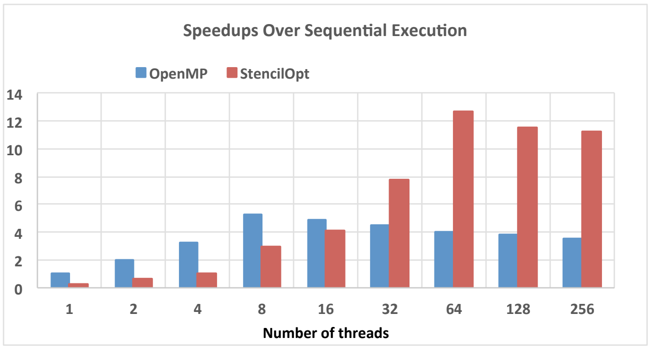


Fig. 10. Impact of thread configurations on 24 time iterations of 256^3 7-point stencil

Fig. 10 compares the performance of the OpenMP and stencilOpt implementations when using different numbers of threads for the 7-point stencil kernel when the array size is 256^3 and when the computation is repeated 24 times. The OpenMP implementations use the default OpenMP policy to schedule these threads onto different hardware cores. On the other hand, the stencilOpt implementations explicitly bind the threads to individual hardware cores to match the actual topology of the stencil when possible, e.g., by parallelizing all data dimensions to form a $n \times n \times n$ topology. From Fig. 10, the best performance by OpenMP is attained when using only 8 out of the 64 cores available on the machine. When more than 8 threads are used, the performance goes down due to network congestions created by the data exchanges among the randomly assigned cores. In contrast, the StencilOpt implementation is able to fully utilize the 64 cores available on the machine to attain close to a factor of 13 speedup over the sequential implementation, compared to a factor of 5 by OpenMP attained when using 8 cores. Although the stencilOpt implementation incurs more significant overhead than the OpenMP baseline, as demonstrated when using fewer than 16 cores, the benefit of better data placement and communication management outweighs the cost when using at least 32 cores.

5.3 Implications of Halo Management

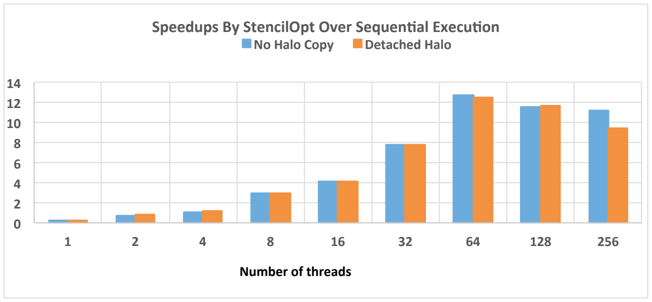


Fig. 11. Impact of halo copying on 24 time iterations of 256^3 7-point stencil

Fig. 11 compares the performance of stencilOpt implementations when using detached halo management vs. using no explicit halo management, which is the default option used in Sections 5.1 and 5.2. In particular, the compiler supports detached halo management by prefetching all the remote memory references into local arrays before using the local copies in the actual computation. The main benefit of pre-fetching halo regions is to enable each core reuse its local copies within a single time step. However, the more remote memory references are involved, the larger memory footprint each thread needs to hold all the local copies, and the large footprints may incur additional cache misses when the memory demand is high. For the 256^3 7-point stencil with 24 time iterations, the performance of stencilOpt either with or without halo copying is similar to each other for a majority of cases except when using 256 threads, where since each hardware core needs to host 4 threads, each thread has a smaller cache allocation which is insufficient to hold all the local copies.

Note that halo copying is often used to simplify the complexity of software development when manually implementing stencil computations on distributed memory platforms, as it is impractical to manually enumerate the different combinations of local and remote references as shown in the automatically generated code in Fig. 6. Using a compiler + runtime combined approach make it unnecessary to hardcode this optimization into the high level source code of the computation, therefore promoting application productivity and portability.

6 Related Work

Existing work has recognized the importance of extending OpenMP compilers to support NUMA architectures to attain high performance. For example, Bircsak et. al [2] investigated user-supplied page migration and data layout directives. Chapman, Patil, and Prabhakar [6] evaluated various ways that OpenMP may be used for performance-oriented programming on ccNUMA Architechtures. Huang et. al [11] worked on enabling locality-aware computing in OpenMP by allowing the developer to manipulate data locations hierarchically. The directives we propose are specially tailored for using NUMA aware OpenMP to support the optimization needs of stencil computations and are therefore not intended as part of the general programming model of OpenMP.

Previous research on optimizing stencil computations have mostly focused on enhancing their data locality and parallelism in concert [7, 8, 13, 20, 22, 23] for multi-core or GPU platforms. These approaches span both manual and automatic code optimizations as well as automated performance prediction and tuning of the optimization configurations (e.g., blocking factors). Bondhugula et al. [4] developed an automated framework that performs parallelization and locality optimizations of stencil codes using the polyhedral model. Liu and Li [15] presented an asynchronous algorithm for reducing synchronization costs and improving locality in stencil computations. Christen et al. [7] presented a strategy for improving locality and exploiting parallelism in a stencil code appearing in a Bio-heat equation targeting the Cell BE and Nvidia GPUs. Our work also includes an automated source-to-source compiler for stencil computations. However, we target many-core NUMA architectures and aim to provide a directive driven framework to support the automated cache management for stencil computations on such architectures.

Datta et al. [9, 10] presented an auto-tuning approach to search for the best optimizations for stencil codes, including their data distribution schemes for NUMA systems, However, their NUMA-aware strategy relies on the first-touch memory policy to perform a page-based distribution. Shaheen et al. [21] focus on spatial-temporal data locality, parallelization, regular memory access, and data-to-core affinity to provide efficient temporal blocking schemes for stencil computations running on ccNUMA systems. Our NUMA-aware decomposition and distribution is driven by topological features of both the stencil arrays and the NUMA hierarchy. In addition to data-to-core affinity, our distribution considers minimizing the overhead by selecting remote access links with the least memory latencies.

Bolosky et. al [3] explored the relations between kernel-based NUMA management policies and multiprocessor memory architectures. Various research efforts have focused on performance evaluation on NUMA architecture using programming models such as OpenMP [6,17] or MPI [14]. Navarro et al. [16] used Locality Communication Graph (LCG) to represent the data locality and used compiler techniques to generate efficient loop iteration/data distribution for NUMA machines. Other research efforts have focused on thread and memory placement [1], data distribution, migration, and replication [5]. This paper present a NUMA study over the latest multi-core NUMA CPUs.

7 Conclusion

This paper presents a pragma-driven special purpose optimizing compiler to automatically convert stencil computations in scientific applications to low-level implementations that invoke a runtime library to explicitly manage the data placement and remote memory references on NUMA many-core architectures. Our automatically optimized code have consistently outperformed OpenMP implementations that use first-touch policies to schedule the computations. We show that through automatically specialized code generation for stencils, our approach provides a combination of good performance, architecture portability, and productivity for developers.

Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-CONF-697198.

References

1. J. Antony, P. P. Janes, and A. P. Rendell. Exploring thread and memory placement on numa architectures: Solaris and linux, ultrasparc/fireplane and opteron/hypertransport. In *High Performance Computing-HiPC 2006*, pages 338–352. Springer, 2006.
2. J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C. A. Nelson, and C. D. Offner. Extending openmp for numa machines. In *Supercomputing, ACM/IEEE 2000 Conference*, pages 48–48. IEEE, 2000.
3. W. J. Bolosky, M. L. Scott, R. P. Fitzgerald, R. J. Fowler, and A. L. Cox. Numa policies and their relation to memory architecture. In *ACM SIGARCH Computer Architecture News*, volume 19, pages 212–221. ACM, 1991.
4. U. Bondhugula, A. Hartono, J. Ramanujan, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI 08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 101–113, New York, NY, USA, 2008.
5. J. M. Bull and C. Johnson. Data distribution, migration and replication on a cc-numa architecture. In *Proceedings of the fourth European workshop on OpenMP*, 2002.
6. B. Chapman, A. Patil, and A. Prabhakar. Performance oriented programming for numa architechtures. In *OpenMP Shared Memory Parallel Programming*, pages 137–154. Springer, 2001.
7. M. Christen, O. Schenk, E. Neufeld, P. Messmer, and H. Burkhart. Parallel data-locality aware stencil computations on modern micro-architectures. In *IPDPS 09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, page 110, Washington, DC, USA, 2009.

8. K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing (SC08)*, 2008.
9. K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 4. IEEE Press, 2008.
10. K. Datta, S. Williams, V. Volkov, J. Carter, L. Oliker, J. Shalf, and K. Yelick. Auto-tuning the 27-point stencil for multicore. In *In Proc. iWAPT2009: The Fourth International Workshop on Automatic Performance Tuning*, 2009.
11. L. Huang, H. Jin, L. Yi, and B. Chapman. Enabling locality-aware computations in openmp. *Scientific Programming*, 18(3):169–181, 2010.
12. A. Kleen. A numa api for linux. *Novel Inc*, 2005.
13. S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. *SIGPLAN Not.*, 42(6):235–244, 2007.
14. S. Li, T. Hoefler, and M. Snir. Numa-aware shared-memory collective communication for mpi. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 85–96. ACM, 2013.
15. L. Liu and Z. Li. Improving parallelism and locality with asynchronous algorithms. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 213–222, New York, NY, USA, 2010. ACM.
16. A. Navarro, E. Zapata, and D. Padua. Compiler techniques for the distribution of data and computation. *Parallel and Distributed Systems, IEEE Transactions on*, 14(6):545–562, 2003.
17. D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, et al. Is data distribution necessary in openmp? In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, page 47. IEEE Computer Society, 2000.
18. OpenMP: Simple, portable, scalable SMP programming. http://www.openmp.org, 2006.
19. D. Quinlan et al. ROSE Compiler Infrastructure. http://www.rosecompiler.org/.
20. G. Rivera and C.-W. Tseng. Tiling optimizations for 3d scientific computations. In *In Supercomputing 00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, Washington, DC, USA, 2000.
21. M. Shaheen and R. Strzodka. Numa aware iterative stencil computations on many-core systems. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 461–473. IEEE, 2012.
22. Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *PLDI 99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, page 215228, New York, NY, USA, 1999.
23. Y. Song, R. Xu, C. Wang, and Z. Li. Data locality enhancement by memory reduction. In *Proceedings of the 15th ACM International Conference on Supercomputing*, Sorrento, Italy, 2001.
24. Q. Yi. POET: A scripting language for applying parameterized source-to-source program transformations. *Software: Practice & Experience*, pages 675–706, May 2012.