# TiDA: High-Level Programming Abstractions for Data Locality Management

Didem Unat[1(✉)], Tan Nguyen[2], Weiqun Zhang[2], Muhammed Nufail Farooqi[1], Burak Bastem[1], George Michelogiannakis[2], Ann Almgren[2], and John Shalf[2]

[1] Koç University, Istanbul, Turkey
dunat@ku.edu.tr
[2] Lawrence Berkeley National Laboratory, Berkeley, CA, USA

**Abstract.** The high energy costs for data movement compared to computation gives paramount importance to data locality management in programs. Managing data locality manually is not a trivial task and also complicates programming. Tiling is a well-known approach that provides both data locality and parallelism in an application. However, there is no standard programming construct to express tiling at the application level. We have developed a multicore programming model, *TiDA*, based on tiling and implemented the model as C++ and Fortran libraries. The proposed programming model has three high level abstractions, *tiles*, *regions* and *tile iterator*. These abstractions in the library hide the details of data decomposition, cache locality optimizations, and memory affinity management in the application. In this paper we unveil the internals of the library and demonstrate the performance and programability advantages of the model on five applications on multiple NUMA nodes. The library achieves up to 2.10x speedup over OpenMP in a single compute node for simple kernels, and up to 22x improvement over a single thread for a more complex combustion proxy application (SMC) on 24 cores. The *MPI+TiDA* implementation of geometric multigrid demonstrates a 30.9 % performance improvement over *MPI+OpenMP* when scaling to 3072 cores (excluding MPI communication overheads, 8.5 % otherwise).

## 1 Introduction

The energy cost for computation is improving at a faster rate than the energy cost of moving data on-chip [28]. However, current multicore programming models offer very little facility to express information about data locality or data movement in the memory hierarchy, while almost all parallel systems contain multiple nonuniform memory access (NUMA) nodes and multiple levels of caches. Current programming models fundamentally assume an abstract machine model, where processing elements within a compute node are equidistant. A data-centric model, on the other hand, can provide programming abstractions that describe how the data is laid out on the system and apply the computation to the data where it resides. Furthermore, as processor chips move towards hundred and even thousand-way parallelism, designs that cluster cores into NUMA regions,


J.M. Kunkel et al. (Eds.): ISC High Performance 2016, LNCS 9697, pp. 116–135, 2016.
DOI: 10.1007/978-3-319-41321-1_7

where cores within a region are cache-coherent but cores across regions are not, are expected to emerge [24]. Taking these architectural trends into the account, locality management will be the key to achieve scalability on the next generation computing systems. In response to these architecture changes, we have developed a tiling based programming model, which preserves data-locality in an application.

Tiling and domain decomposition are both well known methods that enhances both data locality and parallelism. Traditionally tiling, also known as cache blocking, is manually applied to loop iterations in a program. There is a plethora of prior work to automate tiling transformations that focus on iteration space tiling using traditional compiler analysis [17,19,25] and polyhedral compiler analysis for perfectly [1,20,22] and imperfectly nested loops [7,16]. However, there is only limited support for tiling in commercial compilers due to the complexity of generating optimized code without domain-specific knowledge or programmer intervention. Another issue with exclusively compiler-based approaches is that they are agnostic about how the parallel execution of tiles is mapped to the underlying architecture. Loop transformations are carried out independently per nested loop, without respecting data locality in the whole program. Application developers need to have a more direct approach in the programming model to manage memory affinity in a way that can be exploited by both the compiler and the runtime system. We argue that this crucial data locality optimization should be formalized and elevated to a fundamental feature of the programming model given its broad impact on application performance and programmer productivity.

We have developed a tiling based programming model, called TiDA, that provides a multi-language library interface to express parallelism and data locality using a handful of simple programming abstractions. In [26], we introduced the initial design principles of TiDA. In this paper, we describe the underlying abstractions for a generalized tiling-based programming model, present a more mature version TiDA library, unveil its implementation details and present extensive performance analysis on five applications. The over-arching tiling-based programming model enables a natural expression of data decomposition and data layout with *logical tiles* and *regional tiles*, so that an abstract *tile iterator* hides the thread management and mapping of tiles onto the underlying core topology. The implementation of the TiDA API achieves performance portability by isolating architecture specific information to a handful of program parameters, *tile size* and *region size*, and enables metadata to propagate to all loops and functions in the application. We show the effectiveness of the library with five structured grid applications including an advanced combustion proxy application and geometric multi-grid solver. Lastly, both the Fortran and C++ library implementations of TiDA are available online for download at https://bitbucket.org/tidaproject/public-source.

## 2 Programming Model

### 2.1 Data Locality Model

The energy cost of data movement is rapidly becoming a dominant factor, because the energy cost for computation is improving at a faster rate than the energy cost of moving data [28]. In fact, it is projected that with 11 nm technology the energy cost for transporting two floating-point operands for an addition just 5 mm on-chip will be comparable to a simple addition operation itself [24]. The design of on-chip networks poses not only energy but performance constraints as well because contention, latency, and throughput effects can have a significant impact on application execution time [10]. Therefore, given stringent power budgets and the increased cost of data movement, it will no longer be practical to continue to maintain the illusion of a flat and infinitely fast on-chip interconnect. Despite the changes in the abstract machine model of modern multicore architectures, the programming models still have the assumption of uniform distance between the compute units. For example, OpenMP assumes processing units are equidistant to each other and binding threads to the cores are left to the programmer or to the OS. In reality, compute units are not equidistant to each other and there is non-uniform interconnect topology as in Intel's Knights Landing, which has a 2D mesh-based network connecting 72 cores [2]. And yet our current on-chip threading and process models do not offer a natural abstraction for handling this non-uniformity.

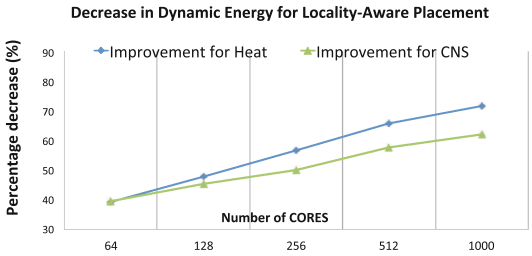


**Fig. 1.** Decrease in dynamic energy consumption for ghost cell exchange for the locality-aware placement compared to random placement of data on the chip.

As a motivating example, we quantify the gains in dynamic energy in on-chip data movement as a result of locality-management. We model the dynamic energy consumption of the ghost zone exchange using an analytical power model [3] for two applications: Heat and CNS, which are explained in Sect. 5. We model efficient direct communication between cores for the ghost zone exchange steps without accessing memory. This direct ghost zone exchange can be created by hardware-managed cache coherence or with explicit data movement for software managed coherence (such as GPUs or local-store architectures). Cores are assigned a $16^3$ tile of double-precision floating point variables each. Locality-aware placement of data reduces the dynamic energy dissipated to complete the

ghost zone exchange by 40–70 % compared to the random placement for the Heat and CNS applications as shown in Fig. 1. The energy gain for the locality-aware placement stems from reducing the average number of hops (communication distance). Reducing the number of hops reduces the number of channels and routers packets traverse which reduces the dynamic energy dissipated. The results on energy dissipation show the importance of correct data placement and the need for locality management support by a programming model. But for conventional threading/process models, both of these features must be handled manually by the programmer – a very unfriendly and non-portable interface.

### 2.2 Programming Abstractions

The primary design goal of TiDA is to provide simple programming abstractions for writing loop oriented code that offers options to describe different data decompositions and abstract away the details of how the data layout changes are implemented in each loop. We present two partitioning abstractions to handle locality as shown in Fig. 2 and an iterator to manipulate them:

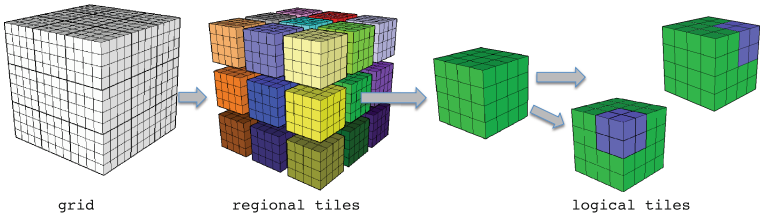


**Fig. 2.** A grid is physically partitioned into regional tiles (regions). Each region is logically partitioned into logical tiles. 27 regions with 8 logical tiles in each are shown.

**Region:** is a physical partitioning of data into *regional tiles* where data within a region is contiguous, but each region is discontiguous with other regions. Such decomposition may introduce *halos* as shown in Fig. 3a, which consist of neighboring cells outside of the local domain at the boundaries that must be updated across computation phases, which will be discussed later. This abstraction is intended to address locality across NUMA nodes or regional coherence domains expected to emerge in exascale node architectures.

**Tile:** is a logical partitioning of data that is expressed in blocking of the iteration space. The *iteration space* is the order in which elements of a data array are visited by the iterations of a nested loop. Whereas repartitioning regional tiles to change working set sizes would require data reorganization to change the tile sizes, logical tiles can shrink the size of working sets to fit within available on-chip memory by changing the blocking factor of the iteration space without requiring data reorganization.

**Tile Iterator:** provides an interface to decouple the loop traversal from the loop body. It can hide complicated traversal orders, parallelization and execution strategies of tiles.

Figure 2 illustrates the partitioning of data and abstractions used for the partitioning. A grid is subdivided into regional tiles and region is locally partitioned into logical tiles.

### 2.3 Parameterization

Determining the optimal number of regions and size of a tile depend on the underlying machine's memory subsystem, the application itself, and other loop optimizations performed by the compiler. Therefore, it is important to support parameterization of the key elements of our tiling abstraction to facilitate performance portability in the programming model, and runtime retuning to support dynamically adaptive codes such as Adaptive Mesh Refinement (AMR).

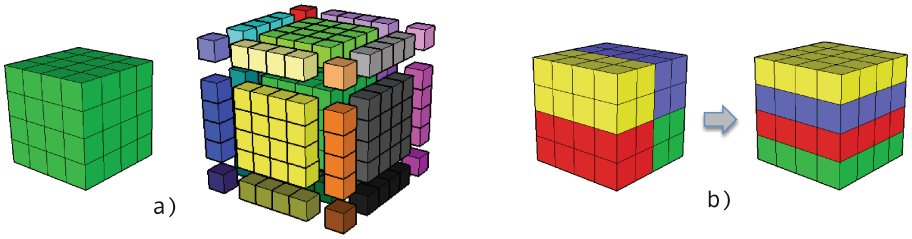


**Fig. 3.** (a) Halo cells of a regional tile is shown, (b) two different geometries of logical tiles in a regional tile. Dynamic tile size allows traversing a region in different orders.

An analytical model would argue that *local* tile sizes that are set at loop-by-loop basis would yield to optimal performance instead of program global tile sizes [27]. The advantage of local tile sizes is that the optimal tile size depends on array usage and loop content because different loops have different working set sizes. On the other hand, changing tile size may bring overhead because metadata needs to be reconstructed when tiling information is changed. Furthermore, different tile sizes can cause some of the threads to access non-local NUMA nodes because thread or memory pinning does not change loop-by-loop basis. Migrating threads to cores that are closer to the source NUMA node is an expensive process and can offset the benefits gained from using different tile sizes. Our programming model allows local tile sizes for logical tiles because logical tiles changes how the data space is viewed and traversed in the computation, can be also used to disable tiling for loops that do not exhibit any cache reuse. For example, in Fig. 3b, a regional tile is divided into logical tiles in two different ways. On the contrary, changing regional tile sizes requires reallocation of the data structures.

## 3 Implementation

### 3.1 Overview

We have developed TiDA as standalone C++ and Fortran libraries to make it easier to integrate into existing code frameworks. The library API provides an

alternative to domain-specific languages or auto-tuning compilers that generate code variants. It also provides an alternative to C++ layout abstractions based on template metaprogramming, which do not interoperate well with other languages. Thus, converting existing applications over to use the TiDA abstractions is not as disruptive as completely rewriting the original application. In fact, the existing naively written loop nests in legacy codes largely remain intact with single line TiDA API calls used to annotate data array allocations as shown in Listing 1.1 and at the entry point to each loop nest or kernel invocation as shown in Listing 1.2. The programming abstractions are not tied to a particular language and can be incorporated into other languages such as Python or Julia.

Currently, the library supports programs operating on block-structured grid applications such as combustion, seismic, weather simulations and image processing. Such applications are generally limited by memory bandwidth on current systems [11,29], and expected to become even more memory bandwidth-constrained on future HPC as memory bandwidth improves more slowly than computational throughput [23]. These applications benefit greatly from tiling to improve cache reuse and from domain decomposition to increase parallelism. Indeed, the novelty of our approach is the generalization of best practices so that a single implementation can be applied for a broad array of codes. Although we are using structured stencils to motivate our demonstration, our generalized interfaces for domain decomposition and tiling are applied pervasively to support parallelization to other classes of algorithms such as dense and sparse linear algebra, particle-in-cell methods, and many others.

### 3.2 TiDA Types

The library provides new data types to embed the programming abstractions into a program. These are:

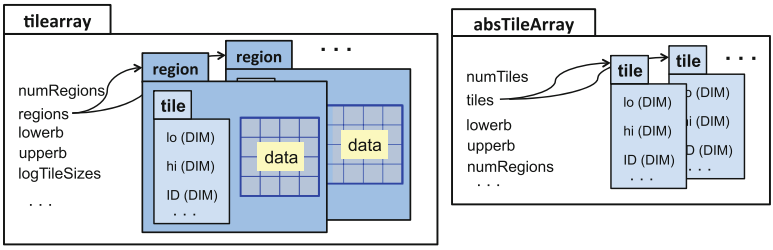


**Fig. 4.** Data structure of `tilearray` and `absTileArray` in TiDA

– `tilearray`: contains data and metadata. A `tilearray` is intended to replace the pure multidimensional array types in the original application that defines the values in a physical domain. The library extends an array with metadata that abstracts away details about data partitioning. The metadata follows the array through the code so that changes in partitioning strategy or mapping do not require any of the computation to be updated. `tilearray` has a pointer to an array of `regions`.

- `region`: represents a region in a `tilearray` and holds the actual data and its iteration space that the data is defined. Typically a TiDA programmer does not directly interact with regional tiles.
- `tile`: merely holds the low and high ends for a rectangular portion of the multidimensional array for logical or regional tiles and does not hold any data. Each `tile` is assigned an ID that uniquely identifies it.
- `absTileArray`: is used to build an abstract structure for `tilearray`s and to create a loop iterator. It defines a multidimensional iteration space through an array of `tile`s. Figure 4 illustrates the interaction of these four data types.
- `tileItr`: is created to iterate over set of tiles in an `absTileArray` or `tilearray`. If a logical tile size is passed to the build method of `tileItr`, then the iterator logically tiles the regions based on the provided tile size. `tileItr` is declared as private to a thread so that each `tileItr` object points to a distinct set of tiles in an `absTileArray`. At the build method, TiDA statically distributes tiles to threads based on thread IDs.

### 3.3 Supporting Parameterization

Selecting the tile size has a great impact on performance and its optimal value depends on the cache size and other loop optimizations such as loop fusion. Thus, being able to configure the tile size brings both performance and productivity benefits. In TiDA, the programmer has the option to specify logical tile size when the data structure is created or through environment variable, called (`TIDA_TILESIZE=tx,ty,tz`), which makes it trivial to tune tile size for an application. A programmer does not need to change any of the solvers as the metadata information propagates to all loops in the code following that array and transparently changes the loop iteration behavior. The geometry of the regions can be set in an environment variable (`TIDA_REGIONS=rx,ry,rz`) as well. The region size is then calculated in the library based on the problem size. For example, on a compute node with four NUMA nodes, choosing region geometry of 1,2,2 or 1,1,4 is expected to yield the best performance.

By default, logical tile size is global and does not change during execution. It is possible to change the global tile size per loop-basis by passing a tile size argument to the tile iterator. TiDA does not allow region size changes during execution except to re-allocate the array with the new layout, making the performance cost transparent to the programmer. In practice the search complexity to find optimal tile size per loop is too large for large-code basis and not desired by the application developers. We generally suggest using this feature only to disable tiling for loops where there is no data reuse because tiling can disrupt hardware prefetchers. The element-wise operations can illustrate the benefit of this feature because operations are performed on independent elements and the loops are highly compute bound. TiDA can be combined with an analytical model [27] to help select the optimal tile size for an application on a given architecture, which is out of scope of this paper.

### 3.4 Tile Boundaries

TiDA provides an interface, `fill_tileboundary()`, to update halos that is needed for structured grid problems. The depth of the halos can be specified in the `build` method when constructing a TiDA array. TiDA updates the halos of the region boundaries in regional tiling. This abstraction is important because it enables codes to migrate to machines with dramatically different memory consistency semantics, such as on software managed memory hierarchies and future systems with regional coherence models. TiDA relies on the programmer to place a call for `fill_tileboundary()` and to handle the inter-node communication (with e.g. MPI). This may introduce an extra copy overhead if the supplementary library is unaware of TiDA. However, this overhead can be eliminated by composing messages directly from a `tileArray`.

### 3.5 Thread and Memory Affinity

Data placement and thread binding play an important role in performance. Without NUMA-aware mapping and execution, the codes scale poorly due to the large memory access latency effects. A programmer using OpenMP can partly control affinity by setting KMP_affinity or GOMP_CPU_AFFINITY. This approach is prone to mistakes and is not portable across platforms. TiDA utilizes the HWLOC tool [15] to query available compute units and their physical numbering to automate thread binding. TiDA binds consecutive thread IDs to consecutive cores in a compact and balanced way. If there are fewer threads than cores, it will distribute them over the cores to increase memory bandwidth but place threads close to each to reduce the halo exchange latency.

In OpenMP programs, it is left to the operating system to bind pages to NUMA domains using the first touch policy. TiDA's region abstraction does not leave the binding to luck and the implementation assigns each region to a NUMA domain. In case a programmer creates only one region, then TiDA performs a parallel initialization to initialize data on NUMA systems, which also implements a first touch page mapping policy. Both thread binding and NUMA-aware mapping are currently static. Future work will look into user-controlled and dynamic affinity management where either thread or data is migrated to adapt the application execution.

## 4 Code Example

The code snippet in Listing 1.1 shows an example to illustrate how a `tilearray` is built in TiDA using the syntax of our Fortran library. Lines 1 and 2 declare variables with type `absTileArray` and `tilearray`. In the next line, `lo` and `hi` are declared as integer vectors defining the low and high ends of the index space of the grid. `tilesizes` and `numregions` are integer vectors for the tile sizes and number of regional tiles, respectively. They are optional arguments to the `_build` method of `absTileArray`. Their values can be read from the environment variables as well. Line 7 builds the metadata for array `A` and `B` with the index

space and chops the space defined by `lo` and `hi` into tiles, and creates an array of `tiles`. Line 8 builds a `tilearray`, allocates its space based on the `absTileArray` and sets the depth of ghost zone. Line 9 builds another `tilearray` with the same structure. Finally, `destroy` in Lines 11–13 free the data structures.

```
1  type(tilearray)  :: A, B
2  type(absTileArray) :: abstractAB
3
4  integer :: lo(2), hi(2)
5  integer :: tilesizes(2), numregions(2)
6  ...
7  abstractAB= absTileArray_build(lo, hi, numregions, tilesizes)
8  A = tilearray_build(abstractAB, nGhosts)
9  B = tilearray_build(abstractAB, nGhosts)
10 ...
11 call destroy(abstractAB)
12 call destroy(A)
13 call destroy(B)
```

**Listing 1.1.** Building TiDA arrays in two dimensions

```
1  type(tileItr) : : ti
2  integer       : : tlo(2), thi(2), reglo(2), reghi(2)
3  integer       : : i, j
4  double precision, pointer : : ptrA(:,:)
5
6  !$OMP PARALLEL PRIVATE(ti, tlo, thi, reglo, reghi, i, j, ptrA)
7
8    ti = tileItr_build(abstractAB)
9    !ti = tileItr_build(abstractAB, logtilesize)
10
11   do while(next_tile(ti))                !<--- Looping over logical tiles
12
13      ptrA =>dataptr(A, ti)               !
14      tlo  = get_lwb(ti)                  ! metadata
15      thi  = get_upb(ti)                  !
16
17      !Option 1: process a tile within a loop
18      do j = tlo(2), thi(2)               !
19         do i = tlo(1), thi(1)            !
20            ptrA(i,j) = compute(i,j)      ! Element
21            ...                           ! loops
22         end do                           !
23      end do                              !
24
25      !Option 2: process a tile within a function
26      reglo = get_lwb(get_region(A, ti))  !
27      reghi = get_upb(get_region(A, ti))  !
28
29      call compute_a_tile(ptrA, tlo, thi, reglo, reghi)
30
31   end do
32 !$OMP END PARALLEL
```

**Listing 1.2.** Operations on TiDA arrays

Listing 1.2 shows an example usage of TiDA. Line 1 declares a tile iterator, `ti`. At line 6, an OpenMP parallel region, spawning multiple threads, is started. In Line 8, `tileItr_build` returns a tile iterator that points to a set of tiles, private to the calling thread, in the tiled array `A`. Line 9 shows a variation of the `tileItr_build` function that creates a tile iterator with a different logical tile size than the one used for constructing `abstractAB`. This feature can be used for implementing function- or loop-specific tile size rather than program global tile size as illustrated in Fig. 3a.

In the do-while statement `next_tile` checks and increments the tile iterator if there are more logical tiles to process. In Line 13 through 15, the code retrieves the `tilearray` metadata. Line 13, `dataptr` returns the pointer to the floating point data for the current tile of the tile iterator `ti` in the tiled array `A`. Depending on the `numregions`, this pointer points to a different location in the grid but all this is hidden behind the TiDA interface. Line 14 and 15 get the lower and upper bounds of the current logical tile in `ti`. Here we demonstrate two different ways to process a `tile`. One way to process a `tile` is using the element loops as shown in line 18 through 23. Element loops iterate over the data points within a `tile`. TiDA does not modify the original loop bodies but introduces tiling loops and new bounds for the element loops. Another way is to process a `tile` within a function containing multiple nested loops as in line 29. In this case, we pass the region sizes to a Fortran subroutine with an explicit-shape array argument for performance reason instead of passing a pointer with no explicit size information. `reglo` and `reghi` on Lines 26 and 27, contain the low and high ends of the region, in which `ptrA` is defined.

Not all loops operate on the interior grid. Some loops may expand to sweep a domain including the ghost zone, such as an initialization loop. For such loops, TiDA provides the `expand_lwb(ti, expansion)` interface to expand lower (and upper) bounds of a tile, where expansion is an integer vector. This function returns expanded bounds of a tile depending on whether the tile resides at the grid boundaries or not.

## 5 Experimental Evaluation

### 5.1 Evaluated Platforms

We use the "Hopper" supercomputer at the National Energy Research Scientific Computing Center (NERSC) for our experiments. A Hopper compute node contains two sockets of 2.1 GHz 12-core Magny-Cours processors. Each socket is comprised of two 6-core chips, totaling four NUMA nodes. The stream bandwidth is 51 GB/s for a Hopper node. All computations use double precision arithmetic.

### 5.2 Performance Evaluation of Single-Mesh Applications

First, we evaluate TiDA performance on single-mesh applications, namely *Heat*, *Wave*, *CNS*, and *SMC*, and compare performance against baseline implementations. The baseline versions perform no tiling and use OpenMP for outermost loop parallelization. For the baseline versions, we set the thread affinities and perform parallel initialization on the NUMA systems. TiDA programs use the programming abstractions described in this paper and use no other manual code optimizations. TiDA implementations use program global tile sizes. We have tuned the logical tile sizes for each application and set the number of regions to number of NUMA nodes for all applications. The characteristics of the applications are listed in Table 1. Figure 5 compares the TiDA library against OpenMP (OMP) with no tiling for a problem size of $256^3$ in double precision on Hopper.

**Table 1.** Characteristics of evaluated applications. #Loops indicate only 3D spatial loops in solvers. Byte/Flop ratios are for unlimited cache.

| | Heat | Wave | CNS | SMC | miniMG |
|---|---|---|---|---|---|
| Stencil | 7-point | 13-point | 27-point | 27-point | 7-point |
| #Halos | 1 | 6 | 4 | 4 | 1 |
| #Loops | 1 | 4 | 14 | 28 | 5/grid level |
| #3D arrays | 2 | 6 | 46 | 157 | 5/grid level |
| Flops/point | 7 | 79 | 1625 | 15K | 20 |
| Byte/flop | 3.43 | 1.14 | 1.32 | 0.82 | 2 |

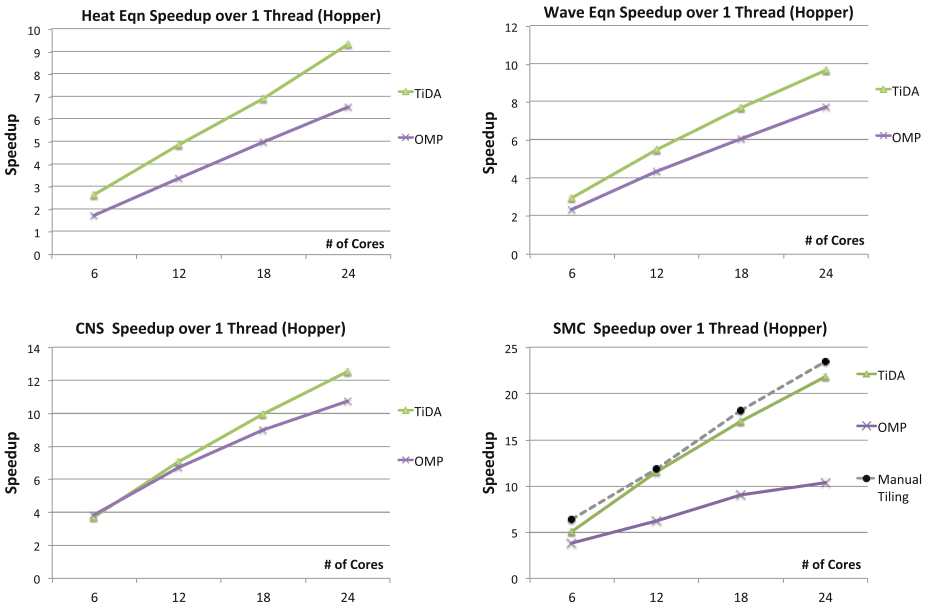


**Fig. 5.** Hopper speedups for *Heat, Wave, CNS and SMC* for problem size of $256^3$.

1. **Heat** solves the heat transfer equation, given a constant heat conduction coefficient and no heat source. The solver iteratively updates a data point using 6 nearest neighbors, requiring three planes to be loaded into the cache for an update. Thus, its working set size is about 1.5 MB $(3 * 8 * 256^2)$, far exceeding the size of the L2 cache on Hopper. Tiling greatly reduces memory traffic: for example, $256 * 16 * 16$ tiles that yield the best performance for Heat also reduces the working size to 90 KB, which falls within the limits of the L2. Smaller tiles come with a trade-off because the halos of each tile must be brought in the cache, leading to an increase in data movement to the point where tiling incurs more traffic than the untiled code.
2. **Wave** studies the constant speed wave equation solved with a third-order Runge-Kutta scheme for time stepping. *Wave* implements a star-shape, 13-

point stencil in a communication avoiding fashion where it expands its halos from 2 to 6 cells so it can compute 3 time steps in a row without exchanging halos. Since a region allocates its domain including its halos separately from other regions, it needs to exchange the deep ghost zone with other regions. Even though for this kernel, filling ghost zone accounts for 10 % of the execution time, TiDA still outperforms the OpenMP implementation.

3. **CNS**, developed by the Exascale Combustion Codesign Center, integrates the compressible Navier-Stokes equations and assumes constant transport properties. This application employs a 27-point stencil kernel. The TiDA improvement over OpenMP is about 17 % on 24 cores. CNS reaches out to four cells in all dimensions when it updates stencil grids in a Runge-Kutta step. However, the updates are performed one derivate at a time and there is reuse only in one of the dimensions in a loop. Because of this property, CNS would benefit from using different logical tile sizes for different loop nests. Moreover, in CNS some loops (4 out of 14) have no reuse and merely stream a number of arrays and perform point-wise operations. In such cases, tiling does not help and performance is ultimately bounded by memory bandwidth.
4. **SMC** [13] is an advanced proxy for the direct numerical combustion codes such as S3D [8]. SMC integrates the multicomponent reacting compressible Navier-Stokes equations with models for chemical species diffusion and kinetics. The dynamical core of SMC uses 8th-order stencil operations to approximate spatial derivatives, converting the system into a large set of ordinary differential equations that are integrated using a third-order, low-storage, TVD Runge-Kutta scheme. The computational cost of the algorithm depends on the number of chemical species and the number of reactions between the species. Our experiments use 9 chemical species.

TiDA is a clear winner for the SMC application, realizing 21.8x the performance on 24 cores as shown in Table 2 and Fig. 5. SMC is particularly challenging for OpenMP and good case study for TiDA because of the high number of data arrays used in the computation. The working set size is very large (about 256 MB for $N = 256$) even for 9 species. The exascale target for SMC is 50 or more chemical species, which will further increase the working set size. As chemical species are added to the simulation, the memory traffic required per Runge-Kutta step increases linearly. Thus, tiling in all dimensions is indispensable for SMC both in current and future machines. We also manually tiled SMC and compared our results on Hopper. The performance of the manual tiling is only 5 % better than TiDA. This indicates that the library introduces a small overhead on the application.

### 5.3 Region and Tile Size Parameters

The results in Fig. 5 in the previous section are obtained by using program global tile sizes and program global number of regions. Even though setting global values for these parameters is easy and provides reasonably good performance, TiDA provides APIs to use local tile sizes or array-specific region sizes. We study

**Table 2.** Hopper running time of TiDA and OpenMP for SMC.

| | Time (sec) TiDA | Time (sec) OpenMP | Speedup TiDA | Speedup OpenMP |
|---|---|---|---|---|
| Baseline | 553.1 | 553.1 | 1.0 | 1.0 |
| 6 threads | 110.2 | 144.9 | 5.0 | 3.8 |
| 12 threads | 48.0 | 89.1 | 11.5 | 6.2 |
| 18 threads | 32.6 | 61.4 | 17.0 | 9.0 |
| 24 threads | 25.4 | 53.3 | 21.8 | 10.4 |

impact of using local parameters on the CNS application because it has 14 loops and accesses 46 three dimensional arrays.

We first study the impact of using local tile sizes on performance. A programmer can set a different logical tile size for a loop nest or for a function when a tile iterator is created. The CNS application is implemented using four main functions, namely *Diffterm, Hypterm, CtoPrim* and *Update*. We created a different tile iterator with a different tile size for each function and measure the execution time. Figure 6 (Left) shows the performance improvement over the program global tile sizes for each function and overall speedup of the application. Two of the functions did not benefit from tile size changes because there is little or no reuse in these functions. *Diffterm* and *Hypterm* have a different optimal tile size than the program global tile size and enjoy 10 % and 6 % speedup, respectively. The overall performance improvement for function-specific tile sizes is 7 %.

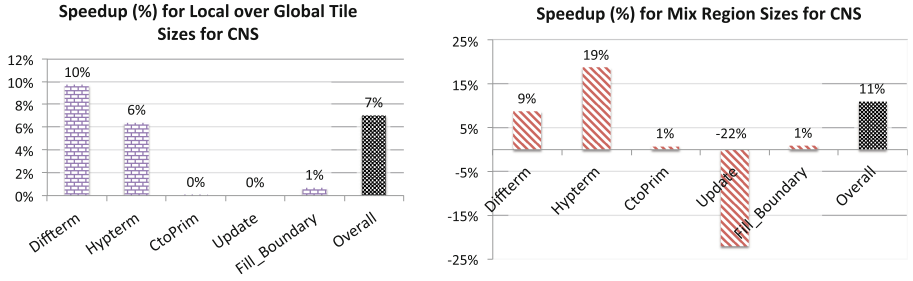


**Fig. 6.** Left: speedup for function-specific tile sizes over program global tile sizes. Right: speedup for array-specific region sizes over program global region sizes.

In TiDA, it is possible to use a different region size per array as long as an iterator uses the same logical tile size to iterate all the relevant arrays. This feature is particularly useful for arrays without halos because one can create smaller regional tiles than there are NUMA domains without paying the extra memory space cost for halos. In an extreme case a regional tile size can be equal to logical tile size. In CNS, 18 of the three dimensional arrays does not

have halo cells. For those we choose a region size that is equal to the logical tile size. Figure 6 (Right) shows the performance improvement for the array-specific region size over program global region size. *Diffterm* and *Hypterm* benefit from array-specific region sizes and give 9 % and 19 % speedup, respectively. Using mix region size lowers the performance of the *Update* function because this function makes references to 36 arrays out of 46 and has very little cache reuse. Nonetheless *Update* accounts less than 10 % of the execution time, overall array-specific region size provides 11 % performance improvement over the program global region size.

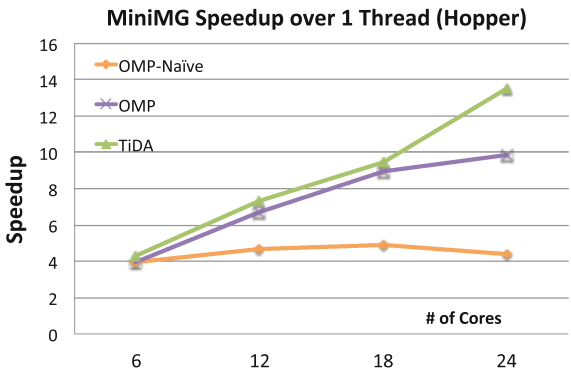


**Fig. 7.** Multigrid strong scaling results on a single Hopper node, finest grid size $256^3$.

### 5.4 Performance Evaluation of Multigrid

**MiniMG** is a compact multigrid application that solves Poisson's equation with periodic boundary conditions. MiniMG iterates V-cycles, which uses Red-Black Gauss Seidel smooths at each grid level. Our original MiniMG implementation employs the data structures provided by Boxlib [30], a software framework for adaptive mesh refinement (AMR). A program written in Boxlib consists of distributed-memory arrays called *multiFabs*, each consisting multiple *Fabs* (Fortran Array Boxes). While *multiFab* can span across different processes, a *Fab* is contiguous in the local memory of a process.

We implemented the multigrid solver in TiDA and compared its performance against OMP. **(i) OMP-naive** parallelizes computations of each *Fab* using all the cores in a compute node without any locality optimizations such as first touch or thread binding. **(ii) OMP** employs a similar parallelization scheme as *OMP-naive* does. However, this variant initializes *Fabs* in parallel to take advantage of NUMA-aware initialization for locality. OpenMP threads execute tiles of a *Fab* in parallel. Similar to *OMP* and *OMP-naive*, there is no parallelism across *Fabs*. **(iii) TiDA** splits *Fabs* into regions (e.g. 4 regions on Hopper) to map each region to a NUMA node, then tiles each region using the *logical* tiling. This variant employs nested OpenMP parallelism. Both regions and tiles in each

region are executed by OpenMP threads in parallel. Program global tile and region sizes are used.

Figure 7 shows the strong-scaling results of the MiniMG variants in a single compute node. For all cases, the solution grid is $256^3$, divided to eight $128^3$ *Fabs*. The poor performance of *OMP-naive* demonstrates the significance of data placement when multiple NUMA domains are present. *TiDA* outperforms other variants on 24 cores and reduces the execution time of *OMP*, a NUMA-aware OpenMP implementation, by up to 37 %. The reason is that *TiDA* does not suffer from the latency overhead caused by spanning the working set across the NUMA nodes. Specifically, *TiDA* overcomes the latency problem by assigning each region to a NUMA node, reducing the number of remote memory accesses. Moreover, given the speculation that an exascale machine will have hundreds of coherence domains, regional tiling will be even more advantageous. The performance of *TiDA* slightly drops on 18 cores due to load imbalance (i.e. assigning 8 *Fabs* to 3 NUMA nodes). In the future, TiDA will mitigate the impact of such load imbalance by supporting tile migration.

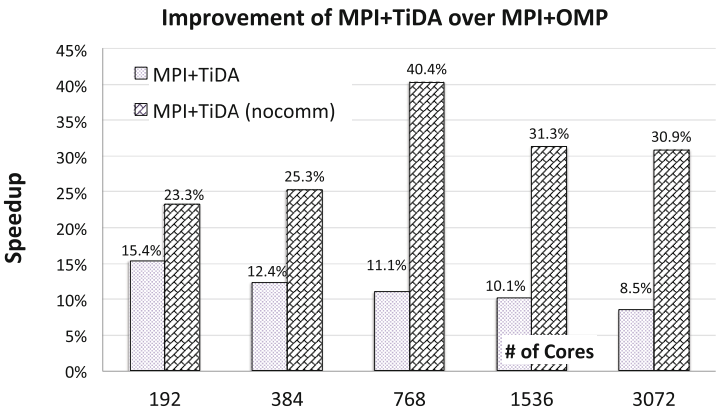


**Fig. 8.** Hybrid programming results for multigrid on Hopper, finest grid size is $1024^3$.

We create hybrid variants that use *MPI+TiDA* and *MPI+OMP* to parallelize the multigrid solver across compute nodes. On-node communication is conducted via sharing memory, whereas off-node communication relies on message passing. Our original multigrid solver implements a truncated V-cycle. Thus, a fixed *Fab* size must be used so that all code variants run the same numerical algorithm. In this study we do not use a pure MPI variant, which employs one MPI process per core because that variant would require very small *Fabs*, reducing the convergence rate of the solution.

Figure 8 shows the strong-scaling study on 128 compute nodes (3072 cores) on Hopper, fixing the finest grid at $1024^3$ and *Fab* size at $128^3$. We can see that *MPI+TiDA* outperforms *MPI+OMP* on up to 3072 cores. However, the performance improvement of *MPI+TiDA* over *MPI+OMP* steadily reduces from

15.4 % on 192 cores to 8.5 % on 3072 cores. This is because the cost of off-node communication grows as the number of cores increases in strong scaling. Indeed on 3072 cores, the communication overhead accounts for 45 % of the execution time, thus the performance benefit of using TiDA becomes less significant. As shown in the same figure, without including the off-node communication time, the performance improvement by *MPI+TiDA* increases from 23.3 % to 30.9 % (going from 192 to 3072 cores). Currently Boxlib is responsible for handling inter-process communication. In the future, we plan to add a runtime support to hide communication overheads by overlapping with computation.

### 5.5 Programming Effort

The primary goal of TiDA is to enable data locality optimizations through abstractions without requiring extensive code changes in the legacy codes. The number of lines of code (#loc) added to existing implementations of the applications is insignificant compared to the size of the programs. For example, for the Heat, we added fewer than 10 lines to build metadata and tilearrays, and added six lines to extract the metadata per nested loop. Moreover, tiling can be performed over a function, which can contain multiple nested loops and no communication phase. Then the #loc can be even smaller because the retrieving tile bounds and pointer to its data can be performed at the function level, significantly reducing the #loc. For example, in the SMC code even though there are 28 nested loops, only four function calls containing those loops needed to be tiled, thus only about 50 lines of TiDA code are added to original 3780 lines of SMC code. When the presented applications ran on another platform, the programmer does not need to modify the source but tune the tile size and region size. Programming effort of using TiDA could be further reduced by elevating the tiling primitives to the level of a language construct (such as an attribute in a Fortran array descriptor) or an embedded directive (extension to OpenMP), but all of TiDA's performance and coding efficiency benefits are available via its library interface.

## 6 Related Work

OpenMP is the most common approach for shared-memory parallelism. OpenMP does not provide a simple abstraction for data decomposition or tiling. The *collapse* clause in OpenMP increases parallelism through flattening the multi-dimensional iteration space into a single dimension but it doesn't implement cache-blocking on the iteration space. The programmer has to introduce nested parallelism to be able to implement tiling however nested parallelism complicates the locality management because it is left to the OS to schedule the newly created threads within a nested region.

Recently, several interfaces and language extensions have emerged to provide data structure and layout abstractions for data locality. Kokkos [12] and Dash [14] support multi-dimensional arrays in C++ and address the intra-chip

data layout changes using C++ meta-programming. GridTool [5] targets multi-stage regular grids that are common in complex weather and climate models and implements the methods with C++ template meta-programming. In both cases, the paradigm is packaged using intense C++ metaprogramming, which is not amenable to other languages such as C, Fortran, Python, etc. Given the installed base of existing code that is written in languages other than C++, TiDA offers language neutral access to these tiling abstractions.

Chen et al. [9] developed tiled MapReduce for large scale data processing with fault tolerance support. Application focus in TiDA is different; it targets structured grid problems, particularly Adaptive Mesh Refinement codes, which are challenging to optimize in large-scale systems. Hierarchically Tiled Arrays (HTA) [6] describes hierarchy and topology of data where computation and communication are represented by overloaded array operations. The array notations hide the cost of temporary arrays and layout transformations, often leading to severe performance penalties, which prevents HTA to be integrated into other parallel libraries. TiDA avoids operator overloading and array notations, and uses abstractions that balance productivity and performance.

Past work has also clearly demonstrated the performance and energy advantages of locality-aware task placement for on-chip data movement. Placements that exploit communication locality and Network on Chip (NoC) topology have been shown to increase effective communication bandwidth by reducing contention on NoCs by 53 % [21], decrease packet latency by 23 % [31], decrease energy by 60 % [18], and provide tighter quality-of-service guarantees compared to locality-agnostic placements [18]. However, past work lacks interaction with the application layer and thus typically resorts to heuristic algorithms or reactive techniques such as process migration [4] using predicted or observed communication graphs and requirements. TiDA improves on past work by generating data-centric and topology-aware mappings based on each application's data structure layout abstracted from the programmer.

## 7 Conclusion

We introduce TiDA as a durable tiling abstraction for data-centric computing. TiDA provides a simple API to describe tile size and data layout and isolates tuning parameters to a single point in the code where the data is instantiated, providing performance portability. The results for five stencil applications show its enhanced scalability potential on HPC systems. Moreover, TiDA's abstractions are forward looking. It supports layouts for alternative cache-coherence mechanisms as massively-parallel chip architectures move towards regional coherence models. Even though we implemented TiDA using Fortran and C++ as the base languages, the abstractions are not tied to these languages and can be implemented in any other languages. If the API is elevated to a language, the metadata retrieval and tuning for tile size and memory layout can be lifted from the programmer to the compiler and runtime, which would further reduce programmer burden.

We are currently developing a runtime system to hide communication overheads between TiDA regions and allow asynchronous execution of tiles. In addition, we plan to extend the current API to target GPU architectures.

**Acknowledgments.** Dr. Unat is supported by the Marie Sklodowska Curie Reintegration Grant 655965 by the European Commission. Authors from KU are supported by the Turkish Science and Technology Research Centre Grant No: 215E285. Authors from LBNL were supported by the SciDAC Program and the Exascale Co-Design Program under the U.S. DOE contract DE-AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. DOE under Contract No. DE-AC02-05CH11231. We would like to acknowledge and thank John Bell and Hakan Memisoglu for their input.

## References

1. PLuTo, A polyhedral automatic parallelizer and locality optimizer for multicores. Software. http://pluto-compiler.sourceforge.net
2. Real World Technologies: Knights Landing Details. http://www.realworldtech.com/knights-landing-details/
3. Balfour, J., Dally, W.J.: Design tradeoffs for tiled CMP on-chip networks. In: Proceedings of the 20th Annual International Conference on Supercomputing, ICS 2006 (2006)
4. Bertozzi, S., Acquaviva, A., Bertozzi, D., Poggiali, A.: Supporting task migration in multi-processor systems-on-chip: a feasibility study. In: Proceedings of Design, Automation and Test in Europe, 2006, DATE 2006, vol. 1, pp. 1–6, March 2006
5. Bianco, M., Cumming, B.: A generic strategy for multi-stage stencils. In: Silva, F., Dutra, I., Santos Costa, V. (eds.) Euro-Par 2014 Parallel Processing. LNCS, vol. 8632, pp. 584–595. Springer, Heidelberg (2014)
6. Bikshandi, G., Guo, J., Hoeflinger, D., Almasi, G., Fraguela, B.B., Garzarán, M.J., Padua, D., von Praun, C.: Programming for parallelism and locality with hierarchically tiled arrays. In: Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPopp, 2006, pp. 48–57. ACM, New York (2006)
7. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. SIGPLAN Not. **3**(6), 101–113 (2008)
8. Chen, J.H., Choudhary, A., de Supinski, B., DeVries, M., Hawkes, E.R., Klasky, S., Liao, W.K., Ma, K.L., Mellor-Crummey, J., Podhorszki, N., Sankaran, R., Shende, S., Yoo, C.S.: Terascale direct numerical simulations of turbulent combustion using S3D. Comput. Sci. Discovery **2**(1), 015001 (2009)
9. Chen, R., Chen, H.: Tiled-mapreduce: efficient and flexible mapreduce processing on multicore with tiling. ACM Trans. Archit. Code Optim. **10**(1), 3:1–3:30 (2013)
10. Das, R., Mutlu, O., Moscibroda, T., Das, C.R.: Application-aware prioritization mechanisms for on-chip networks. In: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO, pp. 280–291 (2009)
11. Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., Yelick, K.: Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: Proceedings of the ACM/IEEE Conference on Supercomputing, SC 2008, pp. 4:1–4:12. IEEE Press, Piscataway (2008)

12. Edwards, H.C., Sunderland, D., Porter, V., Amsler, C., Mish, S.: Manycore performance-portability: Kokkos multidimensional array library. Sci. Program. **20**(2), 89–114 (2012)
13. Emmett, M., Zhang, W., Bell, J.B.: High-order algorithms for compressible reacting flow with complex chemistry. Combust. Theor. Model. **18**(3), 361–387 (2014)
14. Fuchs, T., Fürlinger, K.: Expressing and exploiting multidimensional locality in DASH. In: Proceedings of the SPPEXA Symposium 2016. Lecture Notes in Computational Science and Engineering, Garching, Germany, January 2016
15. Goglin, B.: Managing the topology of heterogeneous cluster nodes with hardware locality (hwloc). In: International Conference on High Performance Computing and Simulation, HPCS 2014, Bologna, Italy, 21–25 July 2014, pp. 74–81 (2014)
16. Hall, M., Chame, J., Chen, C., Shin, J., Rudy, G., Khan, M.M.: Loop transformation recipes for code generation and auto-tuning. In: Gao, G.R., Pollock, L.L., Cavazos, J., Li, X. (eds.) LCPC 2009. LNCS, vol. 5898, pp. 50–64. Springer, Heidelberg (2010)
17. Hartono, A., Baskaran, M.M., Bastoul, C., Cohen, A., Krishnamoorthy, S., Norris, B., Ramanujam, J., Sadayappan, P.: Parametric multi-level tiling of imperfectly nested loops. In: Proceedings of the 23rd International Conference on Supercomputing, ICS 2009, pp. 147–157. ACM, New York (2009)
18. Jingcao, H., Marculescu, R.: Energy-aware mapping for tile-based NoC architectures under performance constraints. In: Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC 2003, pp. 233–239 (2003)
19. Kim, D., Rajopadhye, S.: Parameterized tiling for imperfectly nested loops. Technical report CS-09-101, Department of Computer Science, Colorado State University (2009)
20. Kim, D., Renganarayanan, L., Rostron, D., Rajopadhye, S., Strout, M.M.: Multi-level tiling: M for the price of one. In: Proceedings of the ACM/IEEE Conference on Supercomputing, SC 2007, pp. 51:1–51:12. ACM, New York (2007)
21. Murali, S., De Micheli, G.: Bandwidth-constrained mapping of cores onto NoC architectures. In: Proceedings of the Conference on Design, Automation and Test in Europe - vol. 2, DATE '04, (2004)
22. Renganarayanan, L., Kim, D.G., Rajopadhye, S., Strout, M.M.: Parameterized tiled loops for free. SIGPLAN Not. **42**(6), 405–414 (2007)
23. Rogers, B.M., Krishna, A., Bell, G.B., Ken, V., Jiang, X., Solihin, Y.: Scaling the bandwidth wall: challenges in and avenues for CMP scaling. In: Proceedings of the 36th Annual International Symposium on Computer Architecture, ISCA, pp. 371–382 (2009)
24. Shalf, J., Dosanjh, S., Morrison, J.: Exascale computing technology challenges. In: Palma, J.M.L.M., Daydé, M., Marques, O., Lopes, J.C. (eds.) VECPAR 2010. LNCS, vol. 6449, pp. 1–25. Springer, Heidelberg (2011)
25. Unat, D., Cai, X., Baden, S.B.: Mint: realizing CUDA performance in 3D stencil methods with annotated C. In: Proceedings of the International Conference on Supercomputing, ICS 2011, pp. 214–224. ACM, New York (2011)
26. Unat, D., Chan, C., Zhang, W., Bell, J., Shalf, J.: Tiling as a durable abstraction for parallelism and data locality. In: Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing, 18 November 2013
27. Unat, D., Chan, C., Zhang, W., Williams, S., Bachan, J., Bell, J., Shalf, J.: Exasat: an exascale co-design tool for performance modeling. Int. J. High Perform. Comput. Appl. **29**(2), 209–232 (2015)
28. Unat, D., Shalf, J., Hoefler, T., Schulthess, T., Dubey, A., (eds.) et al.: Programming abstractions for data locality. Technical report (2014)

29. Vega, A., Cabarcas, F., Ramirez, A., Valero, M.: Breaking the bandwidth wall in chip multiprocessors. In: International Conference on Embedded Computer Systems, SAMOS, pp. 255–262 (2011)
30. Zhang, W., Almgren, A., Day, M., Nguyen, T., Shalf, J., Unat, D.: BoxLib with tiling: an AMR software framework. SIAM J. Sci. Comput. (2016)
31. Zhou, W., Zhang, Y., Mao, Z.: An application specific NoC mapping for optimized delay. In: Design and Test of Integrated Systems in Nanoscale Technology, DTIS 2006, 184–188, September 2006