Tiling as a Durable Abstraction for Parallelism and Data Locality

Didem Unat Cy Chan Weiqun Zhang John Bell John Shalf
Lawrence Berkeley National Laboratory
1 Cyclotron Rd, Berkeley, California, USA 94720
dunat, cychan, weiqunzhang, jbbell, jshalf @lbl.gov

***Abstract*—Tiling is a useful loop transformation for expressing parallelism and data locality. Automated tiling transformations that preserve data-locality are increasingly important due to hardware trends towards massive parallelism and the increasing costs of data movement relative to the cost of computing. We propose TiDA as a durable tiling abstraction that centralizes parameterized tiling information within array data types with minimal changes to the source code. The data layout information can be used by the compiler and runtime to automatically manage parallelism, optimize data locality, and schedule tasks intelligently. In this paper, we present the design features and early interface of TiDA along with some preliminary results.**

I. Introduction

There are two main trends in the computer architecture that legitimately concern application developers. First, exponential increases in raw parallelism has replaced nearly two decades of clock rate improvements in a microprocessor. From now on, applications must rely extensively on explicit fine-grained parallelism as a main source of performance improvement. Second, the energy cost of moving data is not improving as fast as the energy required for computation. In the future data movement is expected to become the leading contributor to power consumption and cost of future machines [1]. Whereas current programming environments were designed to assume modest growth in parallelism, uniform costs for communicating, and that FLOPs are most expensive (often at the expense of data movement), the future of computing hinges on preserving data locality (sometimes at the expense of FLOPs) and minimizing data movement.

In order to minimize data movement, applications have to be optimized both for vertical and horizontal data movement. Vertical data movement concerns the management of data through the memory hierarchy from memory to processing units and has to be tuned to increase data reuse in on-chip memory. Horizontal data movement concerns the locality management of non-uniformity in bandwidth and latencies to on-chip memory. The NUMA (non-uniform memory access) issues are already prevalent for on-chip data movement and will be more conspicuous on 1000-core chips, leading to serious performance consequences. To address the programming challenges that result from these trends in computer architecture, programming models play a crucial role in abstracting the complexity for programmers. Current programming models assume equal cost for all data accesses and rely on the cache to virtualize data movement, not reflecting reality in the computer architecture. Thus, application developers need a richer interface to express parallelism and data locality requirements of an algorithm.

Tiling is a loop transformation that is proven to be useful to exploit parallelism and enhance data locality. Despite the long list of literature on this optimization [2]–[9], there is no standard automated solution to transfer tiling information to the compiler and runtime system. Most current methods rely on static loop transformations (usually in the source-to-source translation or in the compiler intermediate representation) and do not allow the runtime system to be involved in decisions about tiling transformations using dynamic data. The status-quo is inadequate for modern adaptive codes such as Adaptive Mesh Refinement (AMR) where crucial information about optimizing data locality are only available at runtime and change during execution. We argue that tiling should be decoupled from the loops and elevated to the programming model for better interaction with compiler and runtime system. A tiling formulation supported as a language construct can expose massive degrees of parallelism through domain decomposition because a tile represents an atomic unit of work – thus making it far easier for the runtime to schedule tasks. Automating the scheduling decisions enables the runtime system to hide the complexity of massive growth in on-chip parallelism from the application developers. Moreover, tiles represent the core concept for data locality because vertical locality can be achieved by hierarchically partitioning the domain and selecting the appropriate tile size at each level. Horizontal locality can be achieved by respecting tile topology and co-locating tiles that share data closer to each other when data is mapped to execution units. This formulation naturally allows multi-level parallelism because coarse-grained parallelism can be expressed across tiles and fine-grained parallelism can be introduced in the forms of vectorization and instruction ordering within a tile.

Although the immediate application of this approach targets data parallel or bulk synchronous stencil operations, atomic nature of the tiling abstraction also makes amenable to future work on asynchronous runtime systems. We envision a programming model of the future that is neither purely bulk synchronous nor purely asynchronous parallel since neither approach is perfect for every situation. Our vision for a future programming model embeds data parallel units within task containers, where the data parallel unit focuses on expression of hierarchy and topology with the tiling abstraction and the task parallel unit focuses on functional partitioning, tile mapping and scheduling.

In this paper, we introduce TiDA as a durable tiling abstraction for data parallelism for the programming model

that we are currently designing. The prototype for TiDA is implemented as a Fortran library to identify the interface requirements. Our ultimate goal is to have tiles as a core concept for parallelism and data locality, and introduce language constructs to manipulate tiles. We anticipate using Fortran as the base language because of its native support for multidimensional arrays, though our designs and principles are not tied to a particular language and can be implemented in any other language, such as C/C++ or Python.

II. Background on Tiling

Even though tiling is not inherently supported in OpenMP, there are ways to implement it. The common misconception is to use the *collapse* clause for tiling. This clause only flattens the multidimensional iteration space into single dimension and the new iteration space is then subdivided among threads. The programmer has to introduce nested thread teams and collapse loops to have the tiling effect on the iteration space. Also, the common usage of OpenMP assumes execution units are equidistant to each other, thus the iteration space partitioning with *omp for* does not express how the data is laid out on the execution units. The new interface[1] in OpenMP 4.0 [10] tries to fix this problem by introducing *places* inspired by the X10 language [11]. Places help the runtime reason about locality for task-oriented parallelism. When creating a team for a parallel region, the programmer can use the thread affinity clause (*proc_bind*) to specify a policy for mapping OpenMP threads to places. However, this still poses a problem for adaptive applications.

Although vendor compilers have limited support for iteration space tiling, there have been a long list of literature focusing on tiled code generation both using traditional [12]–[14] and polyhedral compiler methods for perfectly [15], [16] and imperfectly nested loops [17], [18]. Mint [12] uses compiler directives for annotating loops and generates tiled code that exploits on-chip memory on GPUs. Chill [17] is an autotuning compiler, which can generate tiled code based on the recipes provided by the programmer in a script. Pluto [19] is a polyhedral transformation framework, which can generate tiled code where the tile sizes are fixed at compile-time. Since the tile size has a great impact on performance and should be selected based on memory hierarchy, it is highly desirable to parameterize the tile size and configure it at runtime. PrimeTile generates parametrically tiled code for affine, imperfectly nested loops but it outputs sequential code. PTile [20], a successor of PrimeTile, offers a compile-time polyhedral solution for generating parallel parametric tiling for affine, imperfectly nested loops.

A common issue with code generators is that they are agnostic about the runtime system and how parallel execution of tiles is scheduled on the underlying architecture. Hence, the loop transformations are performed independently per nested loop, without obeying data locality *across* loop nests. Applications need to have a more direct approach for memory affinity management and expression of tiles in a way that can be exploited by an adaptive runtime system, both in selecting tile size and scheduling them.

[1] as of July 2013

To our knowledge, Hierarchically Tiled Arrays (HTAs) [3] is the first attempt in formulating computation in a tiled framework. HTA offers rich semantics for describing hierarchy and topology of data across distributed machines. Parallel computation and communications are represented by overloaded array operations. These array operations hide many details from the programmer but eventually leave the programmer with performance problems such as excessive use of temporary arrays or frequent data layout transformations. HTA has a fixed, user-defined distribution of tiles where the user specifies the mesh topology of the processors and distribution type. We believe the distribution of tiles on processors would be better left to the compiler and runtime because they can improve task placement on the underlying machine topology to minimize data movement. The runtime may also choose to co-schedule tiles on subsets of cores or processors to expose task-level parallelism.

III. Introducing TiDA

TiDA centralizes and parameterizes the tiling information at the data structure. This helps isolate tiling parameters to a single point in the code when the data is instantiated and minimizes the number of places that the code must be changed. TiDA represents only the layout of the leaf tiles and its runtime constructs the tile hierarchy by grouping the tiles in the intermediate levels. The programmer is indifferent about how many levels there are or the tile topology in the intermediate levels because these depend on the organization of memory subsystem. This approach is different from HTA's where the programmer specifies the topology and number of tiles for the intermediate levels of the hierarchy as well as the leaf level.

TiDA assumes that the programmer uses another library (e.g. MPI) or a language (e.g. UPC [23], CoArray Fortran [24]) to domain decompose the entire problem and distribute it across distributed memories prior to tiling on a NUMA node. The rational behind this design choice is to make the true cost of accessing a remote memory explicit and the performance implications more transparent. Moreover, many scientific applications today decompose the application into smaller domains each of which fits into a NUMA node and have an OpenMP parallelism within a node. Thus, adding TiDA to existing implementations is not as disruptive as completely rewriting the original data structures. We offer the means to partition the data within a NUMA node. Figure 1 shows how such domain decomposition may look like on a structured-grid adaptive mesh refinement code, where the entire grid is partitioned into different size boxes at different levels (by BoxLib [25] in this case) and a number of boxes are assigned to a NUMA node. Using TiDA, we further subdivide each box into tiles and address parallelism and locality management within a box.

We prototype TiDA as a library which serves as an intermediate step to language extensions because library interface is not as invasive as changing the type system in a language. Elevating interface to the language will provide the compiler more opportunities to perform code transformations in response to the type information, hence, the next step for TiDA to add the language and compiler support.

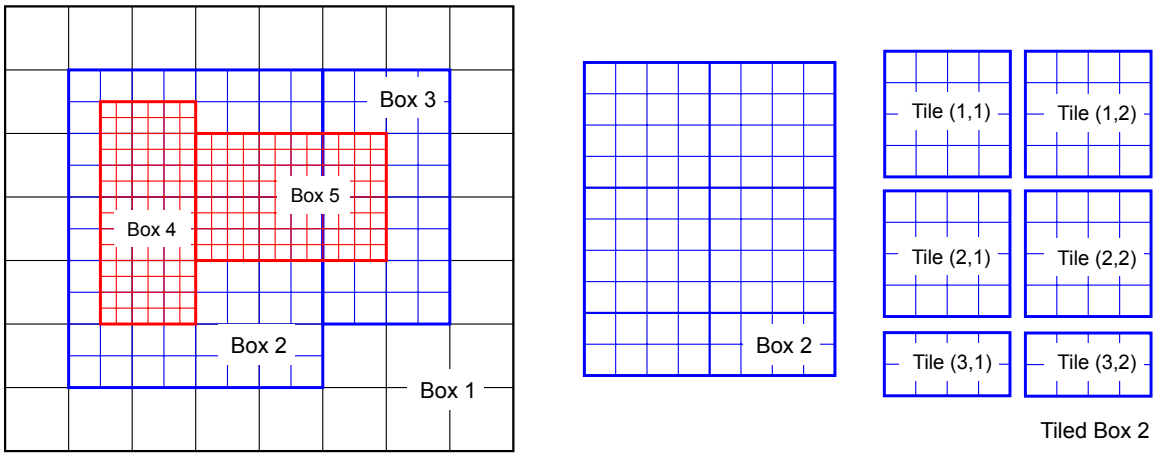

Fig. 1: Box 1 is at level 0, Box 2 and 3 are at level 1, and Box 4 and 5 are at level 2. A number of boxes are assigned to a NUMA node, then each box is subdivide into tiles by TiDA.

A. TiDA Data Structures

TiDA provides four main data structures: two for constructing iteration space and two for actual data.

- `mtile` is the metadata. It represents a rectangular domain in space and does not hold any data. A `mtile` contains the indices of its low end and high end, multidimensional ID, and multidimensional flag indicating whether the tile resides at the boundaries of the box.
- `mtilearray` contains a multidimensional array of `mtiles`. The size of the array in each dimension is the number of `mtiles` in that dimension. `mtilearray` represents the layout of the tiles and their neighboring relation. This information is passed to the compiler and runtime to perform the data placement by respecting data locality.
- `tile` holds data. It contains the `mtile` that it is built on and a pointer to the actual data. A TiDA programmer does not usually deal with a `tile` alone but rather accesses data through a `tilearray`.
- `tilearray` contains a multidimensional array of `tiles` for data and a `mtilearray` to identify the space and layout that data is defined.

B. Tile Boundaries

Both distributed memory and NUMA decomposition of structured grid problems introduce *ghost zone*. The ghost zone consists of neighboring cells outside of the local domain that must be read during computation. The depth of the ghost zone and its shape, thus the access pattern depends on the problem. In many languages including Chapel [26], X10 [11] or Titanium [27], arrays are allocated to accommodate the ghost zone by expanding the domain while iteration on arrays is controlled by the interior domain. However, neither the compiler nor runtime is aware of what ghost cell really implies in these languages. The ghost zone is a read-only shared region during computation and needs to be updated with a synchronization primitive at the end of a timestep. In TiDA, we make the ghost zone part of the data structure, thus its meaning to the compiler and runtime is clear. The depth of the ghost zone can be specified in the `build` method when constructing a TiDA array.

TiDA is responsible for updating the boundaries of tiles within a box and relies on the library or language for inter-box communication. TiDA provides an interface to update the box ghost zone, which introduces an extra copy overhead if the supplementary library or language is unaware of TiDA. This can be eliminated by composing messages directly from TiDA arrays.

C. Memory Layout

The programmer has the options to specify memory layout for tiles when the data is allocated, as shown in Figure 2. One of the options is the *logical* tiling where the `tilearray` is allocated contiguously in the main memory, thus the tiles are logical, only expressed in the iteration space. The second option are *isolated*, which allocates each `tile` contiguously with its ghost zone. This has the overhead of ghost region duplication and increases the memory requirement dramatically for small tile sizes. However, the isolated tiles allows more aggressive loop fusion because synchronization points between tiles can be removed. A programmer does not need to change the program when memory layout option is changed at the TiDA array construction. Computations and communications for different memory layouts are handled implicitly. For example, `fill_boundary()`, which is the interface to update ghost zone, only updates the tiles at the box boundaries when the logical tiling is used, otherwise, it will update the ghost zone of the interior tiles as well.

Another option TiDA will support is to allocate each `tile` *contiguous*ly without duplicating the ghost zone. Even though this option eliminates the redundant ghost regions, it introduces the control complexity due to the possible tile out-of-bound exceptions because a tile may touch neighboring tiles during computation. This layout in theory introduces PGAS on a chip and requires extensive compiler transformation to add the machinery to check whether the data is local to the computing

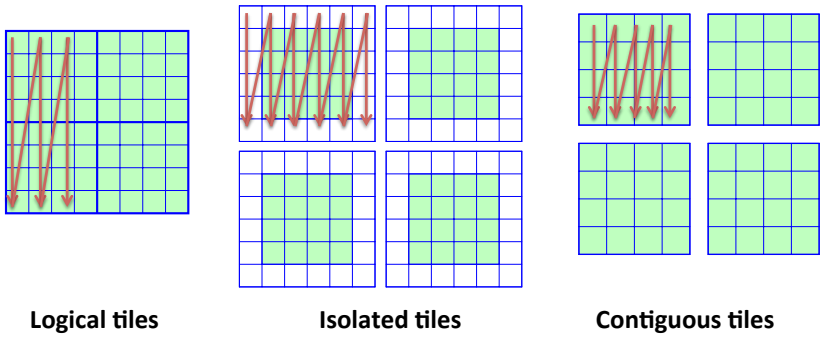

Fig. 2: Memory layout options in `tilearrays`

tile or resides in another tile in the memory. We are still investigating this memory layout option.

IV. Code Example

The code snippet in Listing 1 shows an example to illustrate how a `tilearray` is built in TiDA using the syntax of our Fortran library. Line 1-2 declares two variables with type `mtilearray` and `tilearray`. `lo` and `hi` are integer vectors defining the low end and high end of the index space. `tilesizes` is an integer vector for the tile sizes, which can be set dynamically. Line 8 initializes the `tilearr` with the index space and chops the space defined by `lo` and `hi` into tiles based on the `tilesizes` and creates an array of `mtiles`. Line 9 builds a `tilearray`, allocates its space based on the memory layout provided, sets the depth of ghost zone, and associates the layout of the tiles with the `mtilearray`. Finally, `destroy` in Line 12 and 13 frees the data structures.

```
1  type(mtilearray) :: tilearr
2  type(tilearray)  :: A
3
4  integer :: lo(2)
5  integer :: hi(2)
6  integer :: tilesizes(2)
7  ...
8  call tida_init(tilearr, lo, hi, tilesizes)
9  call tida_build(A, tilearr, numghosts, LOG)
10 ...
11
12 call tida_destroy(tilearr)
13 call tida_destroy(A)
```

Listing 1: Building a TiDA array using `mtilearray` and `tilearray`

Listing 2 shows an example usage of a TiDA array. In Line 5, `ntiles` returns the number of tiles in `tilearr` and the do-loop iterates over them. In Line 7, `dataptr` returns the pointer to the data for a given tile no. Line 9 and 10 get the lower and upper bounds of the tile `tl`. Line 12 and 13 are the elements loops that iterate over the data points within a `tab`.

TiDA does not modify the original loop bodies but introduces tiling loops and new bounds for the element loops. This property brings a great advantage in terms of programming effort because TiDA extensions can be easily added to the existing source codes. At the language level, we would like to decouple the loop traversal mechanism from the loop body and implement the loop body as a lambda function, which will not require any modifications in the loop body. By decoupling, a TiDA compiler can generate different traversal mechanisms for the loops.

```
1  type(tile) :: tl
2  integer    :: tileno, tlo(2), thi(2), i, j
3  double precision, pointer :: ptrA(:,:)
4
5  do tileno=1, ntiles(tilearr)
6
7     ptrA => dataptr(A, tileno)
8     tl = get_tile(tilearr, tileno)
9     tlo = get_lwb(tl)
10    thi = get_upb(tl)
11
12    do j=tlo(2),thi(2)     !element loop 1
13       do i=tlo(1), thi(1) !element loop 2
14          !loop body
15          ptrA(i,j) = do_something(i,j)
16       end do
17    end do
18
19 end do !end of tile loop
```

Listing 2: Operations on TiDA arrays

V. Preliminary Results

To demonstrate the early performance of TiDA, we used the CNS code[2], developed by the Exascale Combustion Co-design Center. CNS is a combustion proxy application that integrates the compressible Navier Stokes equations assuming constant transport. Figure 3 shows the speedup over the serial and untiled implementation for the CNS code. The results are obtained on Intel Westmere[3] using a single socket containing 6 cores and running two hardware threads. TiDA-logical and TiDA-isolated indicate two memory layouts supported in TiDA. Using 12 threads on a 192^3 problem, both the logical and isolated tiles outperform the OpenMP implementation by 20% and 32% perspectively. While OpenMP parallelizes only the outmost loop, TiDA divides the domain into smaller subdomains, each of which fits into cache, thus reduces data movement. Even though the current performance improvements are modest, the results are encouraging.

[2]CNS is available for download at the ExaCT co-design center's website.
[3]Intel Xeon X5680, 12MB cache/socket

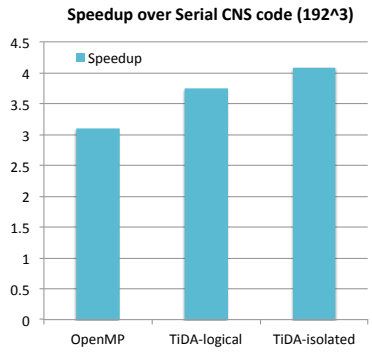

Fig. 3: Speedup over untiled code for TiDA and OpenMP. TiDA-logical indicates column-major logical memory layout and TiDA-isolated indicates tiled-major memory layout for TiDA tiles.

VI. Conclusion

We use tiling as an abstraction for domain decomposition rather than iteration space partitioning and propose data layout description which can be used to express algorithm locality requirements. This description can inform the compiler or runtime to more efficiently map the data to the shared memory subsystem. Tiling if expressed at the array construction level provides a single place to set the data layout without changing the entire code. We are currently in the process of developing the language interface of TiDA, its compiler and runtime support. We will report on the performance data and the progress on the accompanying tools in near future.

Acknowledgment

All authors from Lawrence Berkeley National Laboratory were supported by the Office of Advanced Scientific Computing Research in the Department of Energy Office of Science under contract number DE-AC02-05CH11231. This work is part of the DOE Center for Exascale Simulation of Combustion in Turbulence (ExaCT) and the DOE Co-Design for Exascale (CoDEx) projects.

References

[1] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzen, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick, "ExaScale computing study: Technology challenges in achieving exascale systems," Public Release, DARPA, Tech. Rep., May 2008.

[2] M. Wolfe, "More iteration space tiling," in *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, ser. Supercomputing '89. New York, NY, USA: ACM, 1989, pp. 655–664. [Online]. Available: http://doi.acm.org/10.1145/76263.76337

[3] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzarán, D. Padua, and C. von Praun, "Programming for parallelism and locality with hierarchically tiled arrays," in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '06. New York, NY, USA: ACM, 2006, pp. 48–57. [Online]. Available: http://doi.acm.org/10.1145/1122971.1122981

[4] G. Rivera and C.-W. Tseng, "Tiling optimizations for 3d scientific computations," in *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, ser. Supercomputing '00. Washington, DC, USA: IEEE Computer Society, 2000. [Online]. Available: http://dl.acm.org/citation.cfm?id=370049.370403

[5] S. Carr and K. Kennedy, "Compiler blockability of numerical algorithms," in *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, ser. Supercomputing '92. Los Alamitos, CA, USA: IEEE Computer Society Press, 1992, pp. 114–124. [Online]. Available: http://dl.acm.org/citation.cfm?id=147877.147922

[6] S. Coleman and K. S. McKinley, "Tile size selection using cache organization and data layout," *SIGPLAN Not.*, vol. 30, no. 6, pp. 279–290, Jun. 1995. [Online]. Available: http://doi.acm.org/10.1145/223428.207162

[7] G. Goumas, N. Drosinos, M. Athanasaki, and N. Koziris, "Automatic parallel code generation for tiled nested loops," in *Proceedings of the 2004 ACM symposium on Applied computing*, ser. SAC '04. New York, NY, USA: ACM, 2004, pp. 1412–1419. [Online]. Available: http://doi.acm.org/10.1145/967900.968184

[8] K. Högstedt, L. Carter, and J. Ferrante, "Selecting tile shape for minimal execution time," in *Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures*, ser. SPAA '99. New York, NY, USA: ACM, 1999, pp. 201–211. [Online]. Available: http://doi.acm.org/10.1145/305619.305641

[9] Y. Song and Z. Li, "New tiling techniques to improve cache temporal locality," *SIGPLAN Not.*, vol. 34, no. 5, pp. 215–228, May 1999. [Online]. Available: http://doi.acm.org/10.1145/301631.301668

[10] *OpenMP4.0*, Specifications, July 2013, http://openmp.org/wp/openmp-specifications/.

[11] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," *SIGPLAN Not.*, vol. 40, no. 10, pp. 519–538, Oct. 2005. [Online]. Available: http://doi.acm.org/10.1145/1103845.1094852

[12] D. Unat, X. Cai, and S. B. Baden, "Mint: realizing CUDA performance in 3D stencil methods with annotated C," in *Proceedings of the international conference on Supercomputing*, ser. ICS '11. New York, NY, USA: ACM, 2011, pp. 214–224. [Online]. Available: http://doi.acm.org/10.1145/1995896.1995932

[13] A. Hartono, M. M. Baskaran, C. Bastoul, A. Cohen, S. Krishnamoorthy, B. Norris, J. Ramanujam, and P. Sadayappan, "Parametric multi-level tiling of imperfectly nested loops," in *Proceedings of the 23rd international conference on Supercomputing*, ser. ICS '09. New York, NY, USA: ACM, 2009, pp. 147–157. [Online]. Available: http://doi.acm.org/10.1145/1542275.1542301

[14] D. Kim and S. Rajopadhye, "Parameterized tiling for imperfectly nested loops," in *Technical Report CS-09-101*. Department of Computer Science: Colorado State University, 2009.

[15] L. Renganarayanan, D. Kim, S. Rajopadhye, and M. M. Strout, "Parameterized tiled loops for free," *SIGPLAN Not.*, vol. 42, no. 6, pp. 405–414, Jun. 2007. [Online]. Available: http://doi.acm.org/10.1145/1273442.1250780

[16] D. Kim, L. Renganarayanan, D. Rostron, S. Rajopadhye, and M. M. Strout, "Multi-level tiling: M for the price of one," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, ser. SC '07. New York, NY, USA: ACM, 2007, pp. 51:1–51:12. [Online]. Available: http://doi.acm.org/10.1145/1362622.1362691

[17] M. Hall, J. Chame, C. Chen, J. Shin, G. Rudy, and M. M. Khan, "Loop transformation recipes for code generation and auto-tuning," in *Proceedings of the 22nd international conference on Languages and Compilers for Parallel Computing*, ser. LCPC'09. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 50–64. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-13374-9_4

[18] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," *SIGPLAN Not.*, vol. 43, no. 6, pp. 101–113, Jun. 2008. [Online]. Available: http://doi.acm.org/10.1145/1379022.1375595

[19] *PLuTo*, A polyhedral automatic parallelizer and locality optimizer for multicores. software available at http://pluto-compiler.sourceforge.net.

[20] M. M. Baskaran, A. Hartono, S. Tavarageri, T. Henretty, J. Ramanujam, and P. Sadayappan, "Parameterized tiling revisited," in *Proceedings*

of the 8th annual IEEE/ACM international symposium on Code generation and optimization, ser. CGO '10. New York, NY, USA: ACM, 2010, pp. 200–209. [Online]. Available: http://doi.acm.org/10.1145/1772954.1772983

[21] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: programming the memory hierarchy," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, ser. SC '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1188455.1188543

[22] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: expressing locality and independence with logical regions," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 66:1–66:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389086

[23] UPC Language Specifications, v1.2, UPC Consortium Lawrence Berkeley National Lab Tech Report LBNL-59208, 2005, available at http://upc.lbl.gov/publications/.

[24] R. W. Numrich and J. Reid, "Co-array fortran for parallel programming," *ACM FORTRAN FORUM*, vol. 17, no. 2, pp. 1–31, 1998.

[25] *BoxLib*, Center for Computational Sciences and Engineering, Lawrence Berkeley National Laboratory; software available at https://ccse.lbl.gov/BoxLib/.

[26] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the chapel language," *International Journal of High Performance Computing Applications*, vol. 21, pp. 291–312, 2007.

[27] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken, "Titanium: A high-performance java dialect," in *In ACM*, 1998, pp. 10–11.