SAND2013-9020C

CFD Builder: A Library Builder for Computational Fluid Dynamics

Jagan Jayaraj
Sandia National Laboratories
Albuquerque, NM
jnjayar@sandia.gov

Pei-Hung Lin
Lawrence Livermore National Laboratory
Livermore, CA
phlin@llnl.gov

Paul R. Woodward, Pen-Chung Yew
University of Minnesota
Minneapolis, MN
paul@lcse.umn.edu, yew@cs.umn.edu

***Abstract*—Computational Fluid Dynamics is an important area in scientific computing. The weak scaling of codes is well understood with about two decades of experiences using MPI. As a result, the per-node performance has become very crucial to the overall machine performance. However, despite the use of multi-threading, obtaining good performance at each core is still extremely challenging. The challenges are primarily due to memory bandwidth limitations and difficulties in using the short SIMD engines effectively. This work is about the techniques, and a tool, to improve the in-core performance. Fundamental to the strategy is a hierarchical data layout made of small cubical structures of the problem states that can fit well in the cache hierarchy. The difficulties in computing the spatial derivatives (also called near-neighbor computation in the literature) in a hierarchical data layout are well known, hence, such a data layout has rarely been used in finite difference codes. This work discusses how to program relatively easily for such a hierarchical data layout, the inefficiencies in this programming strategy, and how to overcome its inefficiencies.**

The key technique to eliminate the overheads is called pipeline-for-reuse. It is followed by a storage optimization called maximal array contraction. Both pipeline-for-reuse and maximal array contraction are highly tedious and error-prone. Therefore, we built a source-to-source translator called CFD Builder to automate the transformations using directives. The directive-based approach leverages domain experts' knowledge about the code, and eliminates the need for complex analysis before program transformations. We demonstrated the effectiveness of this approach using three different applications on two different architectures and two different compilers. We see up to 6.92× performance improvement using such an approach. We believe such an approach could enable library and application writers to build efficient CFD libraries.

***Index Terms*—source-to-source; high performance; CFD**

I. Introduction

Computational fluid dynamics (CFD) is an important area of scientific computing used extensively in a variety of fields, including climate modeling, weather prediction, geophysics, aeronautics, astronomy, energy, and defense applications. CFD codes take up much of the computing time on many supercomputers. Speeding them up can improve the utilization of large machines, and can also enable solving more challenging problems.

In CFD, weak scaling (Gustafson's law) across nodes with MPI is common practice. However, achieving good per-node performance has become very challenging with the increasing number of cores on a CPU. The performance gains at the node level can linearly improve the performance of the supercomputer, as long as MPI and the interconnect can handle the increased communication needs [1]. This paper addresses how to improve the per-node performance, and in particular the in-core performance. The other important performance factors such as threading and node level parallelism are already assumed to be programmed.

Techniques to improve the finite-difference methods have been extensively studied under the name of stencil computation. In a stencil computation, the value at a point is the weighted sum of its neighbors. The computation is performed by applying this computational stencil to all the points in a grid. Although the finite-difference codes can be thought of as stencil computation, complex finite difference methods are implemented in a sophisticated fashion by computing partial results. Hence, they no longer have simple computational stencils [2]. This is the primary reason why many of the techniques for stencil computation are not applicable to the finite difference codes discussed in this work.

Researchers have worked on the loop transformations such as loop tiling, loop interchange, loop fusion, and array contraction, for the two most critical factors of in-core performance—data locality and vectorization—for decades (see [3], [4]). The primary objectives of the past research in this area were about the safety (legality) and profitability of a transformation, or a collection of transformations. Examples of profitability analysis for loop tiling include automatically selecting the candidate loops to tile and the optimal tile sizes to use. Finding the most profitable transformation is often NP-complete [4]. Therefore greedy algorithms, heuristics, and runtime search, are primarily employed to find an acceptable solution. The state-of-the-art polyhedral compiler tools such as PoCC [5] employ a brute-force search within a bounded domain to find the best transformations from a very limited set of transformations such as tiling and fusion. However, such approaches are often not scalable to large application codes, and may result in a slow down because the generated code may break the short SIMD vectorization [6]. To take additional transformations such as better SIMD vectorization into consideration will further exacerbate the search time and complexity.

In [2], the efficacy of the above mentioned canonical loop transformations themselves, rather than the profitability algorithms or compiler frameworks implementing them, was evaluated. It finds that even the data reuse from loop fusion does not suffice. The performance improvements occur only with array contraction, in which memory space is reused, as opposed to simply reusing the data. However, all the preceding transformations are necessary to perform array contraction. It is only the combination of all of the above transformations, in conjunction with vectorization, when applied across all the loop nests in the computational region that improves the performance.

Data locality can also be improved by using good data layouts, such as data blocking that conforms to the cache hierarchy, in the application itself. The explicit rearrangement of data into small tiles or blocks will be referred to as data blocking in this work. However, data blocking is not common in Computational Fluid Dynamics (CFD) due to programming complexity. Dynamic data re-layout at a procedure call interface is very effective for linear algebra, but not applicable to CFD. For example in the BLAS implementations, the original matrix in a caller routine is not tiled, but it is copied back and forth to a tiled representation internally in a BLAS routine to improve performance. This approach works when one can perform $O(n^3)$ computations on $O(n^2)$ data, such as those in BLAS, to offset significant copying overhead. It doesn't work in CFD because the amount of computation is directly proportional to the data size. The data transfers become too costly [2], and consequently the CFD codes require a blocked data layout throughout the entire program to be efficient. The enhanced locality from such a blocked data layout facilitates effective vectorization on modern short SIMD engines by making the data assembly required for vectorization very efficient.

Our experience with the Cell processor (see [7]–[9]) forced us to use short aligned vectors for performance, resulting in a new hierarchical data layout and associated code transformations for computation which work exceptionally well on all CPUs. All the three CFD applications evaluated in this paper achieve >20% of the single-precision peak on Intel's Nehalem CPU. Two of the three application codes achieve up to 20% of the single-precision peak on Intel's Sandy Bridge CPU.

Although the resulting code expressions are in a high-level language like Fortran or C, they are very difficult to generate, test, debug, and maintain, because of the nature of the transformations. The transformations need to be performed over the entire computationally intensive region which usually spans multiple procedures. We built a tool called CFD builder to perform the code transformations for CFD library writers and code developers taking advantage of the hierarchical data layout. CFD Builder sidesteps the complex, and often incomplete, program analysis by relying on domain experts' knowledge about their code, and using program directives to direct all of its transformations in a predetermined order. The computationally intensive portion of one of the examples, which is transformed by CFD Builder, consists of 6,082 Fortran lines, across 26 subroutines, after all comments are removed.

The programming complexity of a hierarchical data layout

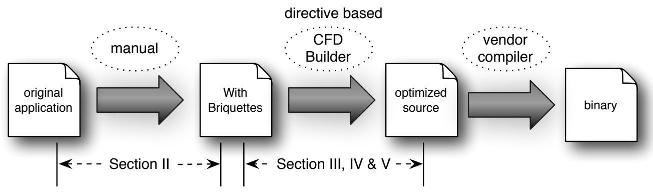


Fig. 1. Overview

is eased by adopting the strategy explained in Section II. The ease of programming comes at the expense of redundant computations. However, CFD Builder completely eliminates the inefficiencies through transformations we call pipelining-for-reuse. This strategy built around blocked data and CFD Builder are evaluated against the four canonical transformations mentioned above, i.e. loop tiling, loop interchange, loop fusion, and array contraction. They outperform even the best combination of canonical transformations, for data locality and vectorization, applied manually by up to 93 percent [2].

Fig. 1 gives an overview of the concepts in this paper. The rest of the paper is organized as follows. Section II provides an overview of the hierarchical data layout. Section III discusses the pipeline-for-reuse transformation, and Section IV discusses how CFD Builder automates pipelining-for-reuse. Section V explains the storage optimization, maximal array contraction, performed by CFD Builder, and Section VI briefly describes the structure of CFD Builder. Section VII describes the evaluation code and experimental setup, and Section VIII discusses the results. Section IX discusses the related work, and Section X contains the concluding remarks.

II. Briquette: A data layout for data locality and vectorization

The primary motivation for the briquettes was to read and write efficiently to the main memory. The problem state of the mesh usually contains many fields. In the Cell processor, the main memory could only be accessed through explicit Direct Memory Access (DMA) calls, and it is more efficient to make fewer calls. The solution was to pack the fields together so that they can be read or written at once. At the same time, the data must be small enough to fit in the on-chip memory. The constraints naturally lead to a data structure where small amounts of data of different fields are packed together. The data structure is called a briquette, and it is defined as the problem state for a tiny cubical region stored contiguously in memory.

The briquette represents the problem state for a small cubical physical region. The individual physical quantities (fields) of the problem state are stored one after the other in a briquette. The briquette is a structure of multidimensional arrays, and can be defined in Fortran as,

```
dimension          bq(nsugar*nsugar,nsugar,nvars)
```

where $nsugar$ corresponds to the number of cells on a side and $nvars$ corresponds to the number of fields in the problem state. The current industrial compilers appear to only

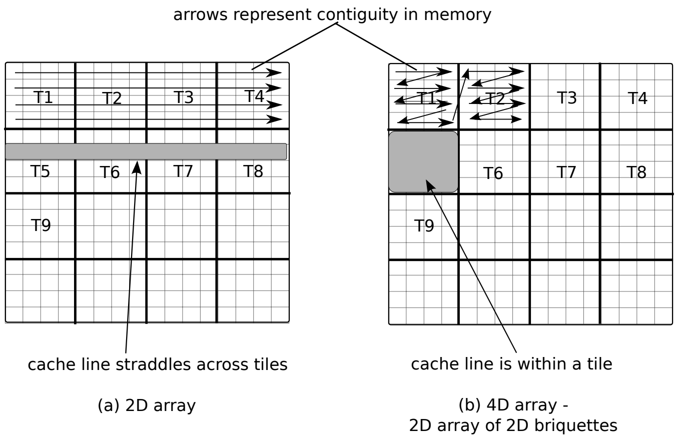

arrows represent contiguity in memory
T1
T2
T3
T4
T5
T6
T7
T8
T9
cache line straddles across tiles
cache line is within a tile
(a) 2D array
(b) 4D array - 2D array of 2D briquettes

Fig. 2. Briquettes vs Tiling

vectorize the inner loop. Therefore the first two dimensions of a briquette are fused in the evaluation codes to maximize the number of iterations in the inner SIMD loop.

Other data structures such as sub-domains and domains are built out of briquettes in a hierarchical fashion. A brick (sub-grid) is a 3-dimensional array of briquettes. In order to overlap communication with computation, each MPI process can update multiple bricks at a time. Woodward et al. update 8 bricks at each process, and they call the collection of eight such bricks as an octo-brick [1]. The bricks and octo-bricks can be defined in Fortran as,

```
dimension     brick(nsugar*nsugar,nsugar,nvars,
                               nbqx, nbqy, nbqz)
dimension  octobrick(nsugar*nsugar,nsugar,nvars,
                               nbqx,nbqy,nbqz, 8)
```

Fig. 2, similar to a figure by Culler et al. [10], illustrates the higher dimensionality of briquettes when compared to tiling. The briquettes are assumed to be squares instead of cubes for illustration. The figure shows how the cache lines are contiguous within each briquette, and not contiguous in a tile.

The benefits of briquette such as coalesced reads and writes, smaller working set, packed operands for SIMD vectorization, efficient data assembly for SIMDization, and support for thread-level parallelism are documented in [2]

Programming with briquettes

Spatial derivatives: Simulating physical phenomena involves solving partial differential equations (PDE). The equations involve quantities which change with respect to time and/or space. Computing how a quantity changes in space is called the spatial derivative of a quantity. Spatial derivatives are essential in CFD where the problem space is discretized into small spatial regions called cells. A spatial derivative at a given point is computed from a set of adjacent cells. Expressing the computation for a domain with briquettes is extremely difficult because the adjacent cells more often will belong to different briquettes making the array indexing very complicated.

Domain decomposition: In a large parallel computation, the problem domain is divided spatially into smaller subdomains to be operated by individual nodes in the system, and this process is called domain decomposition. The neighbor for a cell may be in another sub-domain. The code for spatial derivatives would have been extremely complicated, but the problem is solved by padding each sub-domain with the required neighboring cells from the other subdomains. The padded region is called the ghost region or the boundary region. It is indistinguishable from the real cells for the computation, thereby simplifying programming.

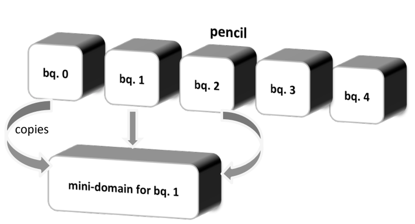
Miniaturization of domain decomposition: The programming complexity of computing spatial derivatives across briquettes can likewise be simplified by padding a briquette with ghost cells. This technique is essentially a miniaturization of the domain decomposition strategy taken to the granularity of briquettes. There are a few differences between the two strategies, though, which are explained here. In domain decomposition, each process typically works a single subdomain. Since the subdomains typically belong to different address spaces, redundant storage must be allocated for the boundary cells in different subdomains. In the miniaturization, each thread of computation works on many briquettes. Since the briquettes for a subdomain belong to a single address space, we can avoid creating redundant storage for all the briquettes. The savings are very crucial for briquettes due to the large ratio of ghost cells to real cells in a briquette. The number of ghost cells on each side of a briquette in a mini-domain is the same as the number of ghost cells on each side of a subdomain.

The briquettes are padded in a separate set of temporary arrays just before their update. In the process, the individual fields of a briquette are copied into separate temporaries to simplify referencing later. This process is called *"unpacking."* The storage for the temporaries is reused by all the briquette updates performed by a thread. However, the computations performed in the ghost regions of a briquette are not reused. CFD Builder has been built to eliminate the redundant computations and copies in the ghost regions between briquettes by reusing the computation.

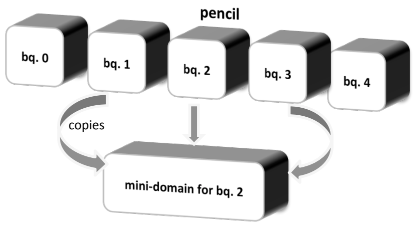
The workspace per thread with briquettes becomes extremely small. It is only 16.59 KB and 19.2 KB per thread for Runge-Kutta advection and PPM advection, respectively. The miniaturized domain decomposition performs 65.61% and 137.31% redundant computations and redundant copies for RK-adv and PPM-adv respectively at a grid resolution of 512-cubed cells II. The redundancy in computation is measured using our Cray-1 style FLOP metric, as described in Section VIII. However, this code with briquettes performs up to 4x faster than the baseline codes without briquettes. In contrast, the linear algebra algorithms do not suffer from any redundant operations or redundant copies due to data blocking.

Below is a list of the sequence of operations to update a briquette:

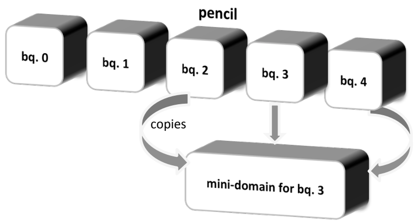
i Unpack the briquette and all the necessary neighboring briquettes into temporary arrays for the individual fields



(a) update for bq. 1



(b) update for bq. 2



(c) update for bq. 3

Fig. 3. Miniaturized domain decomposition

as shown in Fig. 3.

ii Perform the update for the current briquette

iii Pack the results into a briquette and write it back into a new array.

We need a double buffer to hold the old and new values of the problem state. The new results of one pass become the old values for the next pass. The buffers switch roles between passes. However we can eliminate one of the buffers through code transformations as we will see in Section III. The code variant with briquettes and miniaturized domain decomposition will be referred to as ***WB***, an abbreviation for *with briquettes*. The code variant without briquettes will be referred to as ***NB***, an abbreviation for *no briquettes*

Transposing the problem state improves SIMDization in dimensionally split codes by packing the operands. Here transposing is performed efficiently by transposing the briquettes individually. The briquettes in conjunction with the programming strategy deliver up to 4x improvement in performance. The programming strategy, although built around briquettes, works for data blocking too.

III. Pipeline-for-reuse

Despite the advantages, the WB code expression has inefficiencies which are more pronounced for smaller briquette sizes. The inefficiencies can be eliminated through program transformations referred to in this paper as pipelining-for-reuse. This chapter presents an automated mechanism to perform the transformations from source-to-source. The results show up to 2x improvement from pipelining-for-reuse on two different architectures and two different compilers.

The redundant computations in WB can be eliminated by piping the partial results between the briquettes (see III-B for an example of partial results). However, it is desirable to avoid the communication costs for piping the results. Therefore a single processor core is assigned to compute a line of briquettes in the sweep direction. Since the same processor core computes the adjacent briquettes it does not have to communicate with another core for piping the partial results. The results do not actually *"flow"* from one memory location to another. This pipelining is meant to reuse the partial results of computations and it is not aimed at increasing the amount of parallelism. Therefore this pipelining is called as pipelining-for-reuse. Fig. 4 is an illustration of pipeline-for-reuse in action. This section presents pipelining-for-reuse,

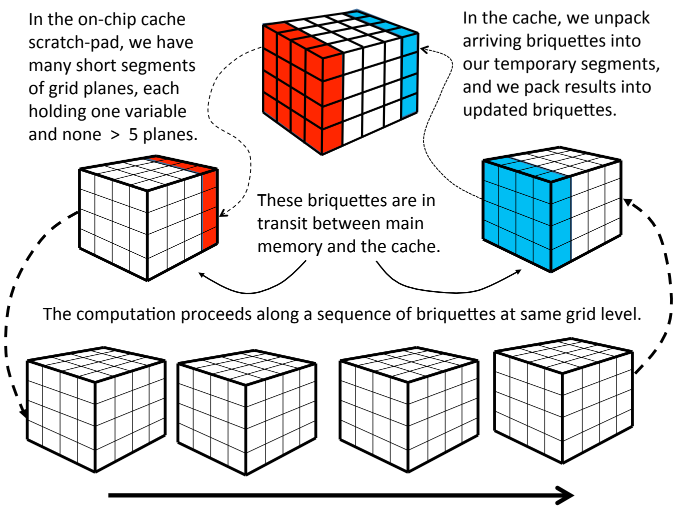

In the on-chip cache scratch-pad, we have many short segments of grid planes, each holding one variable and none > 5 planes.
In the cache, we unpack arriving briquettes into our temporary segments, and we pack results into updated briquettes.
These briquettes are in transit between main memory and the cache.
The computation proceeds along a sequence of briquettes at same grid level.

Fig. 4. Briquettes and Pipelining-for-reuse

the transformations to eliminate redundant computations and copies in the WB code expression.

A. Eliminating redundant copies

Let $rho(nsugar * nsugar, 1 - nbdy : nsugar + nbdy)$ be a Fortran temporary which holds the unpacked density field in WB where $nbdy$ represents the length of the difference stencil in either direction and $nsugar$ represents the number of planes in a briquette. Each briquette needs to be copied and unpacked only once to eliminate redundant copies. This change will overwrite $nsugar$ planes of rho when a newly read briquette is unpacked. It is desirable to overwrite only the spent-up planes which can be accomplished by making rho into a revolving buffer using indirect indices to access them.

Fig. 5 shows how the storage of rho can be reused as the computations proceeds along a pencil of briquettes. Here, $nbdy$ is assumed to be $nsugar$ for the ease of illustration. The abbreviation *"bq."* stands for *briquette* in the picture. On updating briquette 1, the first $nsugar$ planes of rho corresponding to the briquette 0 are stale. They can be overwritten by $nsugar$ planes of rho from briquette 3 when we update briquette 2. This process is repeated for the subsequent briquettes. The planes from briquette $n + 1$ are logically to the right of the planes from briquette n in any given mini-domain. However as we see in Fig. 5(b) and 5(c), the planes from briquettes 3 and 4 do end up physically to the left of planes from briquette 2 in the array rho. Therefore we create a map from the logical planes in a mini-domain for a briquette to the physical planes in the array rho using indirect indexes. The indices are initialized at the beginning of the pencil update. As the computation proceeds, we need to make space for the planes from the new briquettes. Therefore the indices are rotated such that the physical planes to be overwritten are now logically the rightmost planes in the current mini-domain. The rho values from the new briquette can now be unpacked into the freed up space. The computational loops follow the unpacking, as the mini-domain is ready for update.

B. Eliminating redundant computation

The redundant copies to construct rho have been avoided, but the above code expression still performs redundant computations. The partial results computed in the halo region of one mini-domain correspond to the computation in the real regions of other mini-domains. For example, Code 1 shows a simplified computation to be performed at each briquette, in which $rtmp$ is a partial result. $rtmp(:,4)$ for one briquette is same as $rtmp(:,0)$ for the next briquette, and the computation can be reused. The computational loops in WB for a mini-domain have iterations ranging from $nsugar$ to $nsugar + 2 * nbdy$. In other words, at each briquette in a pencil the computational loops perform $nsugar$ and more iterations. Any computational loop which performs more than $nsugar$ iterations is performing redundant computations. The only way to avoid the redundant computations is to make each computational loop perform not more than $nsugar$ iterations for each briquette in a pencil.

Code 1. Partial results

```
do i=0,nsugar
do jk=1,nsugar*nsugar
   rtmp(jk,i) = sqrt(0.6*p(jk,i)*rho(jk,i))
enddo
enddo
do i = 1,nsugar
do jk=1,nsugar*nsugar
   rhonu(jk,i) = 0.5*(rtmp(jk,i-1)+rtmp(jk,i))
enddo
enddo
```

The partial results generated by a computational loop at a given plane are consumed by the subsequent computational loops for the same plane or subsequent planes. As an example, $rtmp$ generated by the first loop nest in Code 1 is consumed by the next loop nest. The partial results need to be buffered till they are consumed by the later loops for the same briquette, or the subsequent briquettes. In contrast, the partial results are never buffered between the mini-domains in WB. The values generated by a later loop are never consumed by a former loop in the computation. The computational loops have a topological order with respect to true dependences. The loops can be grouped into stages of a computational pipeline such that the buffering of partial results mentioned above occurs between the stages. The partial results between two stages remain buffered till the latter stage reaches the appropriate briquette when the results can be consumed. By consuming the results at the earliest possible time, the buffers can be constructed to have the minimal size. The results can be consumed at the earliest only when every computational loop performs as much computation as possible with the briquettes seen so far. It can be achieved by aligning the loops such that the dependence distances between the loops are made as small as possible while preserving the dependences.

The strategy for eliminating the redundant computations in pipeline-for-reuse involves the two steps outlined below:

i Perform loop alignment such that the dependence distances of edges between the computational loops are made as small as possible while still preserving the dependences. Loop alignment is performed to minimize the data buffering between the stages in the computational pipeline.

ii Transform the aligned computational loops to perform at most only $nsugar$ iterations per briquette. if statements are used to prevent iterations which must not be executed. The ifs could have been replaced by loop peeling, but peeling would result in massive code bloat [2].

For a computational loop with an extent from $\{1 - nbdy..nsugar + 2 * nbdy\}$, pipelining-for-reuse will eliminate $2 * nbdy$ redundant iterations for every briquette computed.

IV. Automation of pipelining-for-reuse

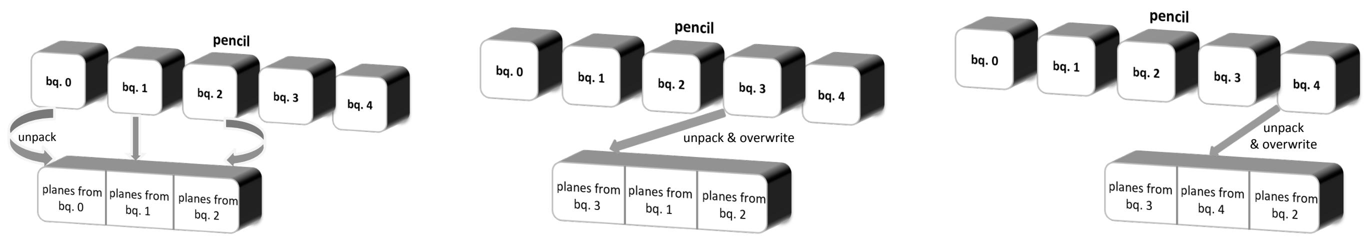
The transformations to eliminate redundant copies and computation are highly error-prone and difficult to implement manually. CFD Builder was built to perform the transformations at the source level automatically with the help of directives listed in Table I. It is built using ANTLR [11] and Java. It reuses the space for the partial results at a very fine granularity of individual planes. CFD Builder also performs double-buffering as part of the pipelining transformation.

The code to update a briquette in WB consists of four major parts: unpack, compute, repack, and write-back. The unpack region constructs the mini-domain to update the briquette. It reads in all the briquettes required to update the current briquette, and copies the individual briquette fields into separate arrays for the ease of programming.

The unpacked arrays are consumed by the compute region which performs the actual computations. The generated results must be repacked into briquettes for the next pass. The repack region performs all the necessary packing and transposing. The write-back region copies the packed briquettes back into the grid to be consumed by the next pass.

A. Eliminate redundant copies

Let $icube$ be the induction variable of loop to update a pencil where each iteration updates a mini-domain. Let $icget$ be the induction variable of a loop inside the $icube$ loop where the required briquettes are read-in to construct a mini-domain. In order to not perform any redundant copies, each iteration of



(a) update for bq. 1 (b) update for bq. 2 (c) update for bq. 3

Fig. 5. Unpacking without redundant copies

$icube$ must read in just a single briquette. It is accomplished by eliminating the $icget$ loop. However, a few briquettes will be left unprocessed by this elimination. Therefore in place of the $icget$ loop, the iteration space of the $icube$ loop is expanded to cover the entire pencil. The iteration space now becomes $\{1-nghostcubes..nbqx+nghostcubes\}$. All the occurrences of $icube$ inside the loop are replaced with $icube-nghostcubes$ where $nghostcubes$ corresponds to the number of ghost cubes. CFD Builder identifies the $icube$ loop and $icget$ loop with the help of cPPM$ PIPELINE and cPPM$ ELIMINATE REDUNDANT ITERATIONS directives, respectively.

B. Double-buffering

We need double buffering to support prefetching data from the global arrays in the main memory to the stack variables in the on-chip memory. In order to double-buffer the briquette reads, the computation must lag behind the prefetch by one briquette. The prefetch has loop carried dependences, of distance one, with the rest of the $icube$ loop body. A prologue region for the first prefetch is avoided by instead delaying the rest of the loop body by one iteration. The $icube$ loop is extended by one more iteration, and the appropriate if conditionals are generated for this loop shifting. The double-buffering provides a place-holder to insert architecture specific prefetching instructions, either hints or commands. CFD Builder identifies the variable names to be double-buffered from the cPPM$ DOUBLEBUFFER directive. Likewise the prefetch region is identified with the help of the cPPM$ PREFETCH BEGIN and cPPM$ PREFETCH END directives placed around the region.

C. Eliminate redundant computations

Inlining: In order to eliminate the redundant computations, all the loops in the computational region need to be merged. However, it is expected to be thousands of lines long as in the ePPM application described in Section VII, and is usually spread across many procedures. Therefore it is necessary to inline the procedures before loop merging. CFD Builder recursively inlines all the procedures in the region. The cPPM$ INLINE directive needs to be placed in front of all the procedure calls in this region to facilitate the inliner. The need for the inline directive can be eliminated if desired.

Alignment: As a first step for pipelining, the loops in the pipeline region—unpack, compute, repack, and write-back loops—must be merged to preserve the data dependencies. However, the loops have fusion-preventing edges between them. The forward differencing and centered differencing equations in CFD codes cause backward dependencies between the loop nests. The dependencies can be eliminated by loop alignment. The algorithms to align loops require the information on alignment thresholds. The alignment threshold for an edge is nothing but the negative of its dependence distance [4]. CFD Builder implements a linear time algorithm for loop alignment by making the three assumptions stated below.

Assumptions:

1) Let i be the induction variable of the loop level to be fused in a loop nest $\mathcal{L}$. After simplification, a subscript of $\mathcal{L}$ with i can only be a simple affine expression i or $i \pm n$ where n is a positive integer. The subscript cannot have any non-linearity, any other loop index variable, and any scaling.
2) There must be at least one subscript i or $i-n$ in $\mathcal{L}$.
3) The loops have a step size of one.

The finite differencing equations typically do not need anything more complicated than the simple expression of the first assumption, for their subscripts of i.

Loop merging and pipelining: The loops pipelined have mismatching ranges i.e. they have a differing number of trip counts. For example, the loops in ePPM belong to 9 mismatching ranges. The text region is estimated to be too large to fit in a typical 256 KB L2 cache alongside the working set. Therefore, CFD Builder generates a single merged loop with appropriate if conditionals instead of index-set splitting or loop peeling [4] [3] to avoid code explosion. It must be noted that CFD Builder assumes that all the loop nests can be merged. Any scalar statement in between the loop nests will become part of the fused loop, and therefore the statement must not alter the program behavior after fusion.

After alignment, CFD Builder adjusts the iterations spaces of some loops so that utmost $nsugar$ iterations of the fused loop are performed per briquette. These transformations described in this paragraph are specific to pipelining WB, and is not part of loop fusion. Pipelining-for-reuse requires indirect indices to access the partial results, and the indices are

Directive	Identifies the
PIPELINE	loop to update a pencil
DOUBLEBUFFER	arrays to be double-buffered
ELIMINATE REDUNDANT ITERATIONS	loop to construct the mini-domains
PREFETCH BEGIN	beginning of the prefetch region
PREFETCH END	end of the prefetch region
INLINE	subroutine call sites to be inlined
LONGITUDINAL LOOP	loops to be merged
REPACK LOOP	loops which assemble the computational results into briquettes

TABLE I
LIST OF DIRECTIVES

	Flops/cell		
	WB-PFR	WB	Redundancy(%)
RK-adv	162.92	379.89	133.18
PPM-adv	276.31	454.60	64.53
ePPM	3218.67	5195.77	61.43

TABLE II
REDUNDANT FLOPS/CELL

identical to the ones generated to eliminate redundant copies. The complexity of both the algorithms presented is $O(n)$, where n is the number of statements, since all the statements are visited not more than once or twice.

Repack and Write-back: The results of the computation are assembled into briquettes. The data assembly loops, also called repack loops, will be pipelined along with the compute loops. When $nbdy$ is not an integer multiple of $nsugar$, the briquettes are only partially constructed at the end of an iteration of the $icube$ loop. In such cases, the output needs to be double-buffered since we do not write back partial briquettes. Double-buffering the output briquette takes more space since one or more planes will be furloughed at any given point in time. Double-buffering the output is avoided here by delaying the briquette construction. The results of the computation are instead held longer in the pipelined temporaries who do not have any idle storage.

The assembled results of the computation like the problem state and the visualization output are written back to the main memory in briquettes. In a steady state condition, a briquette is fully constructed only once every iteration of the loop over briquettes, and not for every loop iteration over the planes. CFD Builder performs code motion and generates appropriate conditional statements to ensure that the briquettes are write-back when they are fully constructed.

V. MEMORY REDUCTION THROUGH MAXIMAL ARRAY CONTRACTION

Storage optimizations are becoming increasingly more essential for performance. Array contraction is a storage optimization which reduces array size while preserving the program semantics. It is referred to as maximal array contraction when the arrays are reduced to the smallest possible size for a given schedule. We have implemented a linear algorithm for maximal array contraction in one dimension under certain assumptions. The contraction in a single dimension is sufficient for finite difference codes in pencilTemps format or WB after pipeline-for-reuse. It is meant not to contract the dimensions corresponding to the inner SIMD loops. The algorithm for array contraction can be extended to multiple dimensions if necessary.

Maximal array contraction takes advantage of the property that the partial results of computations have short lives. For example after pipeline-for-reuse, most of the partial results do not live beyond a few pipeline stages. It means that the circular buffers for the partial results do not have to be as long as the total number of pipeline stages. They can now be made even smaller. Maximal array contraction constructs the smallest possible circular buffer for each partial result. No further memory reduction can be performed on the circular buffers. The reduced workspace may now fit in the on-chip memory, as is the case with all the three example applications on the current Intel architectures. Maximal array contraction results in up to 21% speedup over just pipeline-for-reuse for ePPM when the workspace fits in the L2 cache after contraction.

VI. THE STRUCTURE OF CFD BUILDER

CFD Builder is comprised of two parts: the frontend and backend. The front end is geared towards improving the computational intensity. The backend generates platform specific SIMD and DMA instructions. The back end provides performance portability across the different architectures by generating the appropriate SIMD and DMA instructions where applicable. The precompiler heavily relies on directives to perform the code transformations. We built both the front end and the back end using ANTLR [13].

A. Front end

The front end performs four main operations: inlining, pipelining-for-reuse, maximal array contraction, and prefetcing, as shown in Fig. 6. The pipelining transformation interleaves the computation from the different stages of a briquette update. Most often, the computations are spread across multiple procedures in the input code. Inlining is essential to bring them together before we can perform pipelining. We simplify the liveness analysis for maximal array contraction by retaining the call site names for variables in the inlined code. Although the output from the translator is ugly we try to make it readable by retaining variable names during inlining and not creating new temporary variables for the circular buffers during maximal array contraction. Currently, the front-end only supports Fortran 77, because we have not built parsers for other languages. It should be noted that the intensifying precompiler only transforms the computation region which is relatively smaller than the rest of the code. Rewriting just the computation region in Fortran 77 may be unpleasant, but not impossible. We are considering migrating to other platforms to have access to robust C, C++, and Fortran parsers.

B. Back end

Since utilizing the SIMD engines is key to performance, we built a back end to generate the SIMD instructions for the different architectures. It can generate SSE, AVX, and SPU

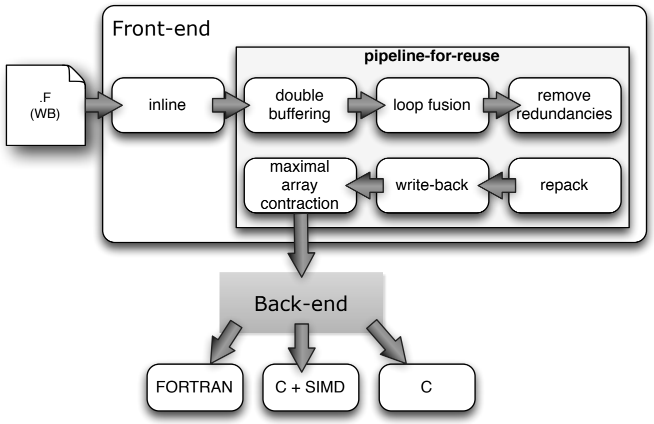


Fig. 6. CFD Builder and Front-End transformations

intrinsics for IBMs Cell. It provides performance portability across the different architectures and native compilers [9]. The back end also generates the DMA instructions for the Cell architecture. It relies on directives to guide the code generation.

The input to CFD Builder is currently WB written in Fortran-77 with a few additional syntactic restrictions. The transformations are invoked through directives in the input code. The use of directives, together with the assumptions made allows CFD Builder to not perform any data dependence analysis. Currently, the incorrect use of the directives or noncompliant input may not be flagged as an error, and may produce incorrect results. For example, it is necessary to merge all the computational loops for pipelining-for-reuse. The statements in between the loop nests will be part of the merged loop as well. Such statements must not alter the behavior of the fused loop, or alter the behavior of the pipelined code. For example if an increment operation occurs in between the loop nests, pipelining-for-reuse will alter its behavior. CFD Builder does not perform strict conformance checks for all the cases currently. However, enhancement of the analysis to report more errors is planned for future work.

VII. Experiments

A. Applications

CFD Builder addresses codes based upon numerical algorithms that use grids which cover a 3-D domain in physical space. CFD codes use grids that are logically uniform or unstructured. Many unstructured grids have very regular local structure. We believe that these codes can still use the techniques that our code translation tool addresses. This can be accomplished by taking each cell of such a grid and chopping it up into, say, 4^3 cells to produce a tiny region of grid logical uniformity.

This study restricts the attention to uniform Cartesian grids in order to show the benefits of our approach in a simpler context. Three codes were used for the demonstration. From our teams own Piecewise Parabolic Method (PPM) gas dynamics codes, we have extracted a code module called PPM advection (it is described in [12]), PPM-adv. The interpolation algorithm in PPM-adv is quite complex, and it involves a wide difference stencil. In this respect, this algorithm is similar to many others that are in general use in the CFD community. This algorithm gives a good illustration of our code transformations while involving relatively little code. From the Cloud Model 1 (CM1) weather code (described in [13]), we have extracted a code module called Runge-Kutte advection (RK-adv) [14].

Both the code modules have been built into full applications by placing them into a code framework similar to our full CFD code for testing. The algorithms are dimensionally split which means that the problem state is updated individually in each of the component directions. The computation is carried out in a sequence of six 1-D passes in a repeated pattern of xyzzyx.

The third application called ePPM is a multifluid gas dynamics code. This multifluid PPM code of Woodward and his collaborators is based upon the original Piecewise-Parabolic Method [15] and PPB advection scheme [16] with various later modifications described in [12] and [17]. It is set up to solve an inertial confinement fusion (ICF) test problem described in [1]. This is a fully functional simulation code, which has scaled on NCSAs new Blue Waters machine to over 730,000 threads running on over 24,000 nodes [1]. After CFD Builder transforms ePPM, the computationally intensive portion consists of just one subroutine with 6121 lines of Fortran (with no comments).

B. Experimental Setup

Since this study is about intra-node performance, the execution times were measured for only the computational regions using CPU_TIME. All other aspects of high-performance computing such as the internode communication and file operations, which are nevertheless important, are not relevant for this study. For this reason, the codes were benchmarked at a single node, but running on all the cores in a node using OpenMP.

The code variants were run on a dual-socket node with quadcore Intel Xeon x5570 (Nehalem) processors @ 2.93 GHz, and on a dual-socket node with eight core Intel Xeon E5-2670 (Sandy Bridge) processors @ 2.6 GHz. Intel Fortran v13 (ifort) and GCC 4.7.1 (GCC) compilers were used on both the architectures. For ifort, the codes were compiled with -O3 and -xSSE4.2 options for Nehalem, and with -O3 and -xAVX compiler options for Sandy Bridge. Similarly for GCC, -O3 and -msse4.2 compiler options were used for Nehalem, and -O3 and -mavx were used for Sandy Bridge.

Simultaneous multi-threading (SMT) was experimented by assigning one OpenMP thread to each virtual processor core, i.e. 16 threads on a Nehalem node and 32 threads on a Sandy Bridge node. It must be noted that even when the cores were not oversubscribed through OpenMP, the hardware SMT was never disabled in the operating system or in the Basic Input Output System (BIOS). For convenience, having one thread per virtual core will be referred to as the SMT mode, and having one thread per core will be referred to as the non-SMT mode. For ifort, the threads were explicitly pinned to cores using the OpenMP environment variable KMP_AFFINITY. However for GCC, better performance was noticed by not pinning the threads to cores, especially for the non-SMT mode. Therefore, the thread pinning was disabled in GCC for all cases. The page size is 4 KB. Both the codes were set up to solve a cubical problem of 512-cubed cells. The execution times reported are averages of 320 values which represent all the computational time steps between two epochs of physical time.

WB was evaluated here using all the three applications: PPM-adv, RK-adv, and ePPM. ePPM is set up to solve a cubical problem of 224-cubed cells, and the execution times reported for ePPM are averages of 256 values. Since this work is about in-core performance, no scaling studies were performed here with multiple MPI ranks, and MPI capability in the framework has been disabled.

VIII. Results

A. Briquette

The workspace per thread of WB is relatively small for all the three codes. The workspace for the advection codes is comparable to the z-pass of NB-circularTemps which has the smallest workspace among all the vectorized passes of the NB code variants. Even for a large code such as ePPM the workspace of WB is only 208.28 KB.

Although not listed here, GCC generated binaries are two to almost four times slower than the ifort generated binaries because GCC does not vectorize most of the computational loops. ifort vectorizes all the computational loops which are critical for performance. Henceforth, all the execution time and performance data in this section will be referring to the ifort results. Please refer to the work by Lin for achieving SIMD performance portability across different compilers [6]. The SMT mode runs almost twice as fast as the non-SMT mode. Although not listed, better performance is seen by not pinning the threads for the non-SMT mode. However, the overall best performance is seen with thread pinning in the SMT mode.

B. Pipeline-for-reuse

1) Improvement over WB: The codes after pipeline-for-reuse, referred to as WB-PFR-lb2ub, consistently perform better than WB for all cases, as seen in Tab. III. The increase in speed for ifort compiled binaries roughly equals the percentage of redundant flops, listed in Tab. reftable:red-flops, eliminated by pipeline-for-reuse. This implies that pipeline-for-reuse works as expected. The performance improvement from GCC is only about 50% or lower for reasons not clearly understood. GCC also runs slower than ifort in all the cases because GCC doesnt vectorize the loops as much as ifort. Like before, the SMT mode improves the performance for all the instances.

Tab. II lists the percentage of redundant computations in WB compared to WB-PFR-lb2ub. The flops are measured Cray 1 style: adds and multiply, reciprocal, sqrts, and exp, count as 1, 3, 5, and 14 flops, respectively. Tab. IV lists the size of the workspace, and Tab. V percentage of peak

		Speed-up from		
		WB	pipelining-for-reuse	Both
PPM-adv	Nehalem	4.08x	1.56x	6.35x
	Sandy Bridge	3.84x	1.80x	6.92x
RK-adv	Nehalem	2.20x	2.03x	4.47x
	Sandy Bridge	2.48x	2.08x	5.16x

TABLE III
Performance improvements

	Workspace / thread (KB)		
	NB	WB	maxArrCtrn
RK-adv	3.26×10^6	16.59	5.81
PPM-adv	9.59×10^6	19.20	6.16
ePPM		229.53	66.29

TABLE IV
Workspace reduction

performance for WB-FPR-lb2nsugar. The size of workspace is same between WB and WB-FPR-lb2ub except for a few arrays which are not double-buffered in WB. The ifort results for the SMT mode are reported here. As seen in Tab. V, the codes run at a high percentage of single-precision floating point peak after pipelining-for-reuse. PPM-adv and RK-adv achieve more than 20% of peak on both the architectures. Even the larger application, ePPM, runs above 10.5% of peak. Pipelining-for-reuse is followed by maximal array contraction in CFD Builder. It must be noted that the performance for ePPM improves further on both Nehalem and Sandy Bridge to 19.81% and 13.45% respectively through the maximal array contraction.

2) Improvement over NB: The previous section shows the improvements from pipelining-for-reuse over WB. However, WB by itself performs significantly better than NB as seen in Tab. III. Therefore the improvement from applying both the transformations in conjunction is put together in Tab. III. Performance improvement of up to 6.92x over NB is seen when the two transformations are combined. The WB-FPR-lb2nsugar code variant is treated as the combination of WB and pipelining-for-reuse. The ifort results in the SMT mode were used for comparison since ifort and SMT mode produce the best results across all the code variants.

We see between 60% and 92% improvement for RK-adv and PPM-adv after pipelining-for-reuse over NB with GCC. The improvements are not as impressive as in Tab. III because GCC vectorizes poorly. However, the results still demonstrate that WB and pipeline-for-reuse are significantly better than NB for both the compilers evaluated here. The briquettes were specifically designed for efficient transposing to aid vectorization. The approach with briquettes and pipelining-for-reuse performs better any combination of the four canonical code transformations mentioned in Introduction.

C. Maximal array contraction

Tab. IV gives the workspace per thread. It must be noted that the workspace per thread includes even the double-buffered arrays involved in prefetch, unpack, and write back operations. Among the three applications, the workspaces of RK-adv and

	Percentage SP peak			
	WB pipelined		maxArrCtrn	
	Nehalem	Sandy Bridge	Nehalem	Sandy Bridge
RK-adv	26.22	20.68	24.40	17.95
PPM-adv	25.15	19.13	26.58	18.29
ePPM	16.49	11.38	19.81	13.45

TABLE V
PERFORMANCE AS A PERCENTAGE OF PEAK

PPM-adv after pipeline-for-reuse are already small enough to fit in the L1 cache that they do not see any benefit from maximal array contraction in 1-D. The performance for the two codes decreases with maximal array contraction most likely because of the increase in the number of indirect indices to address the circular buffers storing partial results which in turn increases the register pressure.

Maximal array contraction in 1-D makes a big difference for ePPM when the workspace is reduced from 229.53 KB, in WB PFR lb2ub, to 66.29 KB per thread. The workspace can now fit in the L2 cache along with the text segment, and up to 21% improvement in performance is seen over WB-PFR-lb2ub even when the number of indirect indices required to index the contract-ed arrays increases. As seen in Tab. V, ePPM now achieves 19.81% and 13.45% of single precision peak on Nehalem and Sandy Bridge respectively. A prefetch buffer in ePPM was purposefully made large to not fit in the L3 cache in order to have a fair comparison between WB and its code variants. However, ePPM can be made even faster by dimensioning the buffer to fit in the L3 cache, and by inserting explicit prefetch intrinsics, mm_prefetch, to read the briquettes. With the enhancements, ePPM was able to achieve about 24% of peak on Nehalem and 17% of peak on Sandy Bridge.

IX. RELATED WORK

A. Data locality

Numerous works have tried to improve the data locality of stencil computation through tiling [3], [4], [18]–[21]. Tiling improves temporal locality by altering the order of computation. One class of tiling techniques improve the poor cache utilization of 3-D PDE solvers by updating a cache resident sub-block fully, and even repeatedly in time [20], before updating another sub-block [18], [19], [21]. They suffer from redundant computations in the ghost region which is more pronounced for smaller sub-blocks. All the other techniques tile the computation without performing redundant computations. However, there is no improvement seen from tiling alone in the dimensionally split codes evaluated in this work [2].

Dimensionally split algorithms by themselves largely ameliorate the strided data accesses in 3-D algorithms when there is sufficient computation performed in each of the 1-D passes. The stencil kernels such as Jacobi and Gauss-Seidel usually studied in the compiler literature do not have enough computation at each grid point. Therefore the techniques developed to address the computations with very low computational intensity are not always best suited for the more complex real applications.

The benefits of data blocking, explicit rearrangement of data, to improve spatial locality are well known [10]. While data blocking is commonly used in linear algebra [22], [23], it is uncommon in CFD. Array indexing to compute spatial derivatives is extremely complicated to program. In comparison, the linear algebra algorithms, such as matrix multiplication and LU decomposition, do not perform near neighbor computation, and hence data blocking is more straightforward in linear algebra.

Array Programming Languages such as Chapel [24] are not hierarchical enough to support cache blocking. The overlapped tiling from HTA library [25] is expensive when applied to smaller blocks, and the redundant operations resulting from computing the spatial derivatives between tiles cannot be eliminated.

Except for the work by Woodward et al. [1], [7]–[9], none of the other works have addressed explicit data blocking at a very small granularity for the finite difference methods. Although the expression of our data layout (briquettes-in-a-brick) in FORTRAN or C is not easy, our programming strategy is designed to simplify the adoption of the data layout as much as we can. The resulting inefficiencies in the form of redundant computations are eliminated by CFD Builder.

B. Frameworks for optimizing libraries

Numerous works have been performed in building and optimizing numerical libraries. They comprise a whole spectrum of tool support for library building. At one end, library generators such as FFTW [26], and SPIRAL [27], custom generate the necessary library routines automatically. At the other end, the library writers can wield the source-to-source transformation infrastructures to optimize their libraries.

Library generators are extremely successful for linear transforms and Basic Linear Algebra Subroutines (BLAS) where a divide-and-conquer (recursion) strategy is very effective. However, the success is not replicated beyond linear algebra. CFD Builder does not provide a library of non-linear operators. Instead, it allows the users to write their own operators i.e. subroutines.

Broadway [28] and Telescoping languages [29] optimize library routines with the help of annotations. However, they miss domain-specific optimization opportunities which cannot be expressed through their annotations, and trade off performance for generality. More recently, works such as CUDA-CHiLL [30] attempt to open up the compiler to library writers. However, our key transformations like pipelining-for-reuse and maximal array contraction cannot be composed in their polyhedral framework.

CFD Builder sidesteps the analysis with the help of directives and by exploiting the structure of WB. A directive based approach is not uncommon. Directives are widely used for parallelization (eg: OpenMP) in high performance computing, and vector directives have been around for a long time. More recently, annotations, directives, and scripting interfaces, have been used to invoke many more transformations [30], [31].

The source-to-source transformation infrastructures like ROSE, CETUS, and POET, allow for building custom compiler transformations [31]–[33]. CFD Builder has been built using ANTLR, but it can also be implemented in the above compiler infrastructures.

X. Conclusion

This paper is about improving the in-core performance of finite differencing methods. The combination of the briquette data layout, pipelining-for-reuse and maximal array contraction, increase the computational intensity of CFD algorithms. Since the briquette is a modification of data blocking for improved coalesced memory accesses, it is one of the best strategies for spatial locality. The briquette, by design, supports data alignment and SIMD vectorization. Pipelining-for-reuse exploits temporal locality. The approach with briquettes, WB, and pipelining-for-reuse, appears to be a more comprehensive solution for performance than the existing solutions. However, the code expression necessitated by this combination of techniques is highly unreadable and unmaintainable. We have built a developmental source-to-source precompiler to perform the tedious code transformations. We have demonstrated its utility by achieving a high percentage of processor peak on three different applications. We intend CFD Builder to be a productivity tool for building efficient libraries in CFD.

Acknowledgment

This work has been supported through grant CNS-0708822 from the NSF and by the DoE through contracts from the Los Alamos National Lab and Sandia National Lab. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energys National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

[1] P. R. Woodward *et al.*, "Simulating turbulent mixing from richtmyer-meshkov and rayleigh-taylor instabilities in converging geometries using moving cartesian grids," in *Proceedings of the 2012 Nuclear Explosives Code Development Conference (NECDC 2012)*, Feb 2013.

[2] J. Jayaraj, "A strategy for high performance in computational fluid dynamics," Ph.D. dissertation, University of Minnesota, Twin Cities, August 2013.

[3] M. J. Wolfe, *High Performance Compilers for Parallel Computing*, C. Shanklin and L. Ortega, Eds. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[4] K. Kennedy and J. R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[5] L.-N. Pouchet *et al.*, "Loop transformations: convexity, pruning and optimization," *SIGPLAN Not.*, vol. 46, no. 1, pp. 549–562, Jan. 2011.

[6] P.-H. Lin, "Performance portability strategies for computational fluid dynamics (CFD) applications on HPC systems," Ph.D. dissertation, University of Minnesota, Twin Cities, June 2013.

[7] P. R. Woodward, J. Jayaraj, P.-H. Lin, and P.-C. Yew, "Moving scientific codes to multicore microprocessor cpus," *Computing in Science and Engineering*, vol. 10, no. 6, pp. 16–25, 2008.

[8] P. R. Woodward, J. Jayaraj, P.-H. Lin, and W. Dai, "First experience of compressible gas dynamics simulation on the los alamos roadrunner machine," *Concurr. Comput. : Pract. Exper.*, vol. 21, no. 17, pp. 2160–2175, Dec. 2009.

[9] P. R. Woodward *et al.*, "Boosting the performance of computational fluid dynamics codes for interactive supercomputing," *Procedia Computer Science*, vol. 1, no. 1, pp. 2055 – 2064, 2010, iCCS 2010.

[10] D. E. Culler, A. Gupta, and J. P. Singh, *Parallel Computer Architecture: A Hardware/Software Approach*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[11] T. Parr, *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

[12] P. R. Woodward, *The PPM Compressible Gas Dynamics Scheme*. Cambridge Univ., 2006.

[13] G. H. Bryan and J. M. Fritsch, "A benchmark simulation for moist nonhydrostatic numerical models," *Monthly Weather Review*, vol. 130, no. 12, pp. 2917–2928, 2013/10/14 2002.

[14] L. J. Wicker and W. C. Skamarock, "Time-splitting methods for elastic models using forward time schemes," *Monthly Weather Review*, vol. 130, no. 8, pp. 2088–2097, 2013/10/14 2002.

[15] P. Colella and P. R. Woodward, "The piecewise parabolic method (ppm) for gas-dynamical simulations," *Journal of Computational Physics*, vol. 54, no. 1, pp. 174 – 201, 1984.

[16] P. R. Woodward, "Numerical methods for astrophysicists," *Astrophysical Radiation Hydrodynamics*, pp. 245–326, 1986. [Online]. Available: www.lcse.umn.edu/PPMlogo

[17] P. R. Woodward, F. Herwig, and P.-H. Lin, "3d hydrodynamic simulations of entrainment at the top boundary of shell convection in stellar interiors. methods and convergence study," *Astrophysical Journal*, 2013, submitted to. [Online]. Available: http://arxiv.org/pdf/1307.3821.pdf

[18] G. Rivera and C.-W. Tseng, "Tiling optimizations for 3d scientific computations," in *Supercomputing, ACM/IEEE 2000 Conference*, 2000, pp. 32–32.

[19] A. W. Lim, S.-W. Liao, and M. S. Lam, "Blocking and array contraction across arbitrarily nested loops using affine partitioning," *SIGPLAN Not.*, vol. 36, no. 7, pp. 103–112, Jun. 2001.

[20] S. Sellappa and S. Chatterjee, "Cache-efficient multigrid algorithms," *Int. J. High Perform. Comput. Appl.*, vol. 18, no. 1, pp. 115–133, Feb. 2004.

[21] Z. Li and Y. Song, "Automatic tiling of iterative stencil loops," *ACM Trans. Program. Lang. Syst.*, vol. 26, no. 6, pp. 975–1028, Nov. 2004.

[22] A. McKellar and E. G. Coffman, "The organization of matrices and matrix operations in a paged multiprogramming environment," *Commun. ACM*, vol. 12, no. 3, pp. 153–165, 1969.

[23] E. Angerson *et al.*, "Lapack: A portable linear algebra library for high-performance computers," in *Supercomputing '90., Proceedings of*, 1990, pp. 2–11.

[24] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, Aug. 2007.

[25] J. Guo, G. Bikshandi, B. B. Fraguela, and D. Padua, "Writing productive stencil codes with overlapped tiling," *Concurr. Comput. : Pract. Exper.*, vol. 21, no. 1, pp. 25–39, Jan. 2009.

[26] M. Frigo and S. Johnson, "Fftw: an adaptive software architecture for the fft," in *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, vol. 3, 1998, pp. 1381–1384 vol.3.

[27] M. Püschel *et al.*, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, pp. 232– 275, 2005.

[28] S. Guyer and C. Lin, "Broadway: A compiler for exploiting the domain-specific semantics of software libraries," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 342–357, 2005.

[29] A. Chauhan, C. McCosh, K. Kennedy, and R. Hanson, "Automatic type-driven library generation for telescoping languages," in *Supercomputing, 2003 ACM/IEEE Conference*, 2003, pp. 51–51.

[30] G. Rudy *et al.*, "A programming language interface to describe transformations and code generation," in *Proceedings of the 23rd international conference on Languages and compilers for parallel computing*, ser. LCPC'10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 136–150.

[31] Q. Yi, "Poet: a scripting language for applying parameterized source-to-source program transformations," *Softw. Pract. Exper.*, vol. 42, no. 6, pp. 675–706, Jun. 2012.

[32] D. J. Quinlan, “Rose: Compiler support for object-oriented frameworks.” *Parallel Processing Letters*, vol. 10, no. 2/3, pp. 215–226, 2000.

[33] C. Dave *et al.*, “Cetus: A source-to-source compiler infrastructure for multicores,” *Computer*, vol. 42, no. 12, pp. 36–42, 2009.