Int J Parallel Prog (2012) 40:4–24
DOI 10.1007/s10766-011-0182-5

Data Layout Transformation Exploiting Memory-Level Parallelism in Structured Grid Many-Core Applications

I-Jui Sung · Nasser Anssari · John A. Stratton · Wen-Mei W. Hwu

Received: 1 February 2011 / Accepted: 13 July 2011 / Published online: 18 August 2011

Abstract We present automatic data layout transformation as an effective compiler performance optimization for memory-bound structured grid applications. Structured grid applications include stencil codes and other code structures using a dense, regular grid as the primary data structure. Fluid dynamics and heat distribution, which both solve partial differential equations on a discretized representation of space, are representative of many important structured grid applications. Using the information available through variable-length array syntax, standardized in C99 and other modern languages, we enable automatic data layout transformations for structured grid codes with dynamically allocated arrays. We also present how a tool can guide these transformations to statically choose a good layout given a model of the memory system, using a modern GPU as an example. A transformed layout that distributes concurrent memory requests among parallel memory system components provides substantial speedup for structured grid applications by improving their achieved memory-level parallelism. Even with the overhead of more complex address calculations, we observe up to 10.94X speedup over the original layout, and a 1.16X performance gain in the worst case.

Keywords GPU · Parallel programming · Data layout transformation

1 Introduction

Structured grid applications [2] are a class of applications that calculate grid cell values on a regular (structured in general) 2D, 3D or higher dimensional grid. Each output point is computed as a function of itself and its nearest neighbors, potentially with

I.-J. Sung (✉) · N. Anssari · J. A. Stratton · W.-M. W. Hwu
University of Illinois at Urbana-Champaign, Urbana, IL, USA
e-mail: sung10@illinois.edu

patterns more general than a fixed stencil. Examples of structured grid applications include fluid dynamics and heat distribution that iteratively solve partial differential equations (PDEs) on dense multidimensional arrays. For parallelizing such applications, the most common approach is spatial partitioning of grid cell computations into fixed-size portions, usually in the shape of planes or cuboids, and assigning the resulting portions to parallel workers e.g. Pthreads, MPI ranks, or OpenMP `parallel for` loops.

However, the underlying memory hierarchy may not interact in the most efficient way with a given decomposition of the problem; due to the constantly increasing disparity between DRAM and processor speeds [18], modern massively parallel systems employ wider DRAM bursts and a high degree of memory interleaving to create sufficient off-chip memory bandwidth to supply operands to the numerous processing elements.

Unlike CPU-based systems in which a DRAM burst usually corresponds to a cache line fill, massively parallel systems such as GPUs form a DRAM burst from vectorized memory accesses. This can either be done by hardware from concurrent threads in the same wavefront (also known as memory coalescing in CUDA terms) or by the programmer (such as the short-vector loads in CUDA and OpenCL). In both cases, it is important to have concurrent accesses bearing desired memory address bit patterns in terms of memory access vectorization. Intuitively, this can be addressed by loop transformations to achieve unit-strided access in the inner loop. However, for arrays of structures, it is necessary to employ data layout transformations, such as dimension permutation, to achieve vectorization [11] or reduce coherence overhead [12].

A less explored direction is the parallelism among memory controllers, and interleaved DRAM banks, which plays an increasingly important role in system performance. In massively parallel systems, the interconnect between DRAM channels and processors decodes address bit fields to decide the corresponding channel and memory bank numbers from a memory request [3]. Given that a fixed subset of the address bits is used to spread accesses across parallel memory channels and banks, achieving high bandwidth requires concurrently serviced accesses to have varying values in those address bit fields. To exploit this level of memory-level parallelism (MLP) in structured grid applications, precise control must be exercised over how multidimensional index expressions map each index field to address bit fields. It is not generally possible without data layout transformation or hardware approaches [15] to shuffle address bit fields such that concurrent memory requests can be both well-vectorized and routed to different memory channels and banks.

Unfortunately, the full details of a memory hierarchy are often too obscure or complex for typical application programmers to adapt their programs to them. Even for the exceptional cases where the programmer does know how to transform the data layout to fit the memory system, performing the transformation manually is tedious, results in less readable code, and must be repeated every time a new platform is targeted.

To alleviate these problems, we present a formulation to enable automatic data layout transformation for structured grid applications through a monotonic dataflow analysis compiler pass. We augment this formulation with a static tool to help the programmer prune the search space of target layouts for the one that best fits the underlying memory system. Using a modern GPU as an example, we show how to guide the decision process with an analytical model of the memory system based on detailed microbenchmarking that leverages insights into the execution model of a GPU.

Currently, programming languages such as C and FORTRAN rigidly define the layout of multidimensional arrays, allowing usages relying on the default layout such as addressing logically adjacent elements through hard-coded pointer arithmetic. Therefore, programmers opting to use automatic transformations on arrays must be subject to more stringent interfaces that insulate the source code from changes in the layout. However, implementing arrays of transformable layouts using a new language data type or C++-style classes both complicates the language and may incur undesirable overheads for accessing the most performance-critical data structure of the application. To make data layout transformation feasible in the context of a language derived from C++, we rely on two assumptions. First, the declaration, allocation, and access of multidimensional arrays should follow C99-style variable length array (VLA) syntax, Second, the programmer must adhere to FORTRAN-style subscripted array accesses, as any assumptions on the relation between addresses across multiple array elements may not hold after layout transformation.

Figure 1 depicts our procedure for data layout transformation, using a modern GPU memory system as an example. The input is a kernel in which arrays are declared and accessed in a restricted form of variable-length arrays, clearly denoting the size of each array dimension, with array access restricted to FORTRAN-like form. Knowledge of the execution model is then used to determine the relationships and ranges of array indices likely to be concurrently requested by the kernel. For each array of interest, an optimization problem is formulated and solved based on the estimated number of concurrent instances of each array index with distinct values, with the solution determining the desired layout. A code generation pass emits the transformed code with array access expressions converted to flattened array accesses using transformed layouts.

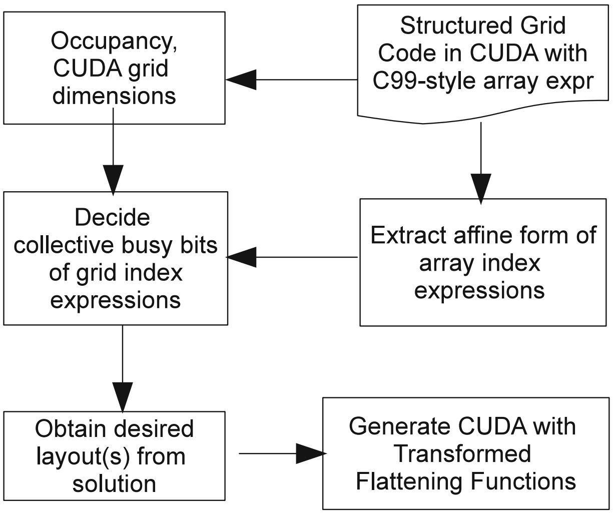
Occupancy, CUDA grid dimensions
Structured Grid Code in CUDA with C99-style array expr
Decide collective busy bits of grid index expressions
Extract affine form of array index expressions
Obtain desired layout(s) from solution
Generate CUDA with Transformed Flattening Functions

Fig. 1 Data layout transforms for structured grid codes

The rest of this paper explains our methodology and results in detail. Section 2 provides an overview of iterative PDE solvers. Section 3 discusses related work in data layout transformations. Section 4 formulates the offset calculation of array-accessing statements, and defines the data layout transformations we consider using this formulation. Section 5 discusses how we obtained a memory address interleaving scheme for DRAM controllers through micro-benchmarking, and derived an optimized layout from the program and execution model. Section 6 presents our experimental results. Finally, Section 7 concludes.

2 Common Access Patterns of PDE Solvers on Structured Grids

Although there are many numerical methods that deal with PDEs, there are only a few data access patterns among the most prevalent methods solving these problems on structured grids. The structured grid often comes from discretizing physical space with Finite Difference Methods [20] or Finite Volume Methods [7], while solutions based on Finite Element Methods [20] often result in irregular meshes.

Many numerical methods solve PDEs through discretization and linearization. The linearized PDE is then solved as a large, sparse linear system [9]. For large problems, direct-solution methods are often not viable: practical approaches are almost exclusively iterative-convergence methods.

Iterative techniques like the Jacobi and Gauss-Seidel methods (including those with Successive Overrelaxation) are often used as important building blocks for more advanced solvers like multigrid [6]. Both techniques are instances of stencil codes, whose stencils can be expressed as a weighted sum of a cell and its nearest neighbors in the grid. The major difference in terms of access patterns is that Gauss-Seidel methods typically apply cell updates in an alternating checkerboard style. Adjacent elements are never updated in the same sweep; two separate, serialized sweeps over the red and black cells perform one whole iteration update.

The lattice-Boltzmann method (LBM) [25], a particle-based method mainly used in computational fluid dynamics problems, was recently extended as a general PDE solver [32]. LBM is also an iterative method applied to structured grids. The cell update rules for LBM are divided into two stages that update multiple grid cell properties (i.e. distribution functions of particles close to different edges or surfaces of the grid cell.) The intra-cell stage (called collide) and inter-cell stage (called stream) combined perform one iteration's update [24]. The stream stage accesses the nearest neighbors of the current cell, while the collide stage's inputs are entirely local to the current cell. Since there is no data reuse within an update iteration across cells, techniques that aim at reducing memory accesses such as shared memory tiling for the GPU are less useful. Hence, LBM is considered memory bandwidth-bound [27].

3 Related Work

Since stencil codes and LBM applications are often memory bandwidth-bound, many approaches have focused on enhancing the memory system performance for these applications. However, most of these approaches focus on increasing the cached reuse of data loaded from memory. For traditional cache-based memory hierarchies, most methods do so by transforming the traversal order of array elements by loop tiling at the cache line size [26,28].

Stencil codes are a subset of structured grid applications that have been studied extensively, and optimized for locality on many platforms, including the GPU platform we address in this paper [5]. Because there is no traditional cache or direct control over the relative execution order of threads, most GPU-specific transformations for stencil codes aim to enhance reuse of shared data across neighboring cells using a pipeline-like approach, e.g. Datta et al. [5].

Such methods, which improve how efficiently data is used or reused in the on-chip cache of the system, are not always applicable or sufficient. For example, the LBM does not contain any data reuse within a single timestep [24], and some stencil codes with heavy reuse may still be performance bound by off-chip bandwidth even after reuse is exploited. Applications in such situations could potentially still gain significant performance improvement by using MLP-oriented optimizations.

Data layout transformations [17] have been primarily used for improving cache locality and localizing memory accesses in nonuniform memory architectures and clusters. Anderson et al. [1] employed data layout transformations for shared-memory multiprocessor systems to make the data accessed by the same processor contiguous in the address space. Lu et al.'s recent work [16] applied data layout transformation for cache locality in NUCA (non-uniform cache architecture) chip multiprocessors to keep memory accesses localized in processor-local L2 cache which have non-uniform access cost, which results in similar formulation of earlier works for clusters [13]. This work presents data layout transformations to aid hardware memory vectorization and reduce DRAM bank conflicts, with the target architecture being many-core processors connected to on-chip multi-channel memory controllers through on-chip interconnects with uniform access cost from each core to each memory channel, e.g. a GPU.

In terms of the underlying DRAM memory model, most of the work described above only considered the latency of hitting or missing in the data cache, or the latency of local versus remote memory accesses for clusters and NUCA. However, for a massively parallel system, balancing DRAM traffic across controllers can be important. Datta et al. [5] took into consideration the affinity of DRAM controllers and processor cores in NUMA architectures using an affinity-aware memory allocator. To our knowledge, there is no software-based approach to balancing workloads for the multi-channel, interleaved DRAM controllers employed in modern parallel architectures.

For GPUs, we know of no previous work applying data layout transformation to structured-grid codes other than for gaining unit-strided accesses [11,27], which helps vectorizing memory accesses into DRAM bursts (i.e. coalescing). Our automated data layout transformations further exploit concurrency in multi-channel and interleaved DRAM bank organizations. For managing off-chip memory bandwidth, Baskaran et al. [4] proposed an approach based on loop tiling using the polyhedral model framework, effectively assigning thread indices so that access patterns can be better coalesced. However, their approach considered the data layout as part of the constraints and hence is not to further improve the memory access efficiency on strided accesses commonly seen in the LBM and red-black Gauss-Seidel methods codes.

Various works [10,21,22,29] proposed hardware optimizations for DRAM controllers towards uniform access latency and fairness, some parallelism aware [21,22]. All of them focused on scheduling DRAM requests under fixed workloads, without considering how the workloads themselves could potentially adapt to the memory system. Also, such approaches only balance among memory banks under the same controller, while several controllers are often present in modern systems.

4 Data Layout Transformations for Structured Grid C Code

For structured grid codes, transforming the bit patterns of effective addresses of concurrent grid access expressions for the underlying memory hierarchy can be achieved by transforming linearization functions calculating grid elements' offsets from index expressions for each dimension and the size of each dimension. This effectively transforms the data layout.

We first present a formalization of arrays, layouts, and layout transformations that define the required information as well as semantics. To conduct data layout transformation, we collect the necessary information through variable-length array syntax, a recently standardized feature of the C language, that enables FORTRAN-style index expressions for arrays of all kinds, including those whose size is not statically known. The extra information contained in these declarations and accesses are essential to performing robust data layout transformation.

4.1 Grids and Flattening Functions

Definition 1 An n-dimensional array G is characterized by an index space that is a convex, rectangular subspace of $\mathbb{N}^n$.

An array element is identified by a vector of integers called an *index vector*. Without loss of generality, for the index vector $\mathbf{I}$ of an array element, $I_i \in [0, Dim_i)$ where $Dim_i \in \mathbb{N}$, $Dim_i > 0$ is the i-th element of the *dimension vector* of G.

Definition 2 An injective function FF: $\mathbb{N}^n \rightarrow \mathbb{N}$ is a flattening function for an n-dimensional array G if this function is defined for all valid array element index vectors.

A flattening function defines a linearization of coordinates of elements in G. When the resulting integer is interpreted as the offset for addressing an element from the beginning of the memory space reserved for the array, then this flattening function defines the memory layout of the array. We require FF to be injective: it should map every valid index vector to a unique value. An FF f explicitly forbids a many-to-one mapping, and thus f^{-1} is defined and $f^{-1}(f(\mathbf{I})) = \mathbf{I}$ for a valid index vector $\mathbf{I}$. With these restrictions, a flattening function uniquely defines a memory layout and vice-versa; we use these terms interchangeably in the remaining text.

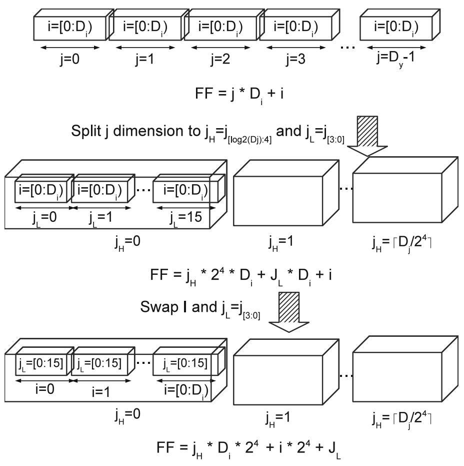

Fig. 2 An example of layout transformation

To permute the address bit pattern derived by an FF, we can transform the default Row-Major Layout (RML) flattening function by adapting the two primitive transformations proposed by Anderson et al. [1] that are analogous to well-known loop transformations:

Strip-mining: Split dimension i into T-sized tiles, $0 \leq T < D_i$. This transformation creates a new index vector $\mathbf{I}'$ and a new dimension vector $\mathbf{D}'$, which are inputs to the transformed FF. $\mathbf{I}'$ and $\mathbf{D}'$ are created by dividing I_i into $I_h,\ I_l$ and D_i into $D_h,\ D_l$, where $I_h = \lfloor I_i/T \rfloor,\ I_l = I_i \bmod T$ and $D_h = \lceil D_i/T \rceil,\ D_l = T$. Intuitively, strip-mining splits a dimension into two adjacent dimensions. When the original dimension size is not a multiple of the strip size, padding is introduced at the last strip.

Permutation: Permute the index vector and corresponding dimension vector.

Figure 2 shows a layout tiling example that transforms an access to array A$[D_j][D_i]$ from A[j][i], i.e. RML_A, to A$[\mathtt{j}_{\log 2(\mathtt{D_i}):4}][\mathtt{i}][\mathtt{j}_{3:0}]$. First the dimension j is split into j_H and j_L without actually changing the order of elements in memory, only padding the grid to some multiple of $2^4 \times D_i$ elements. Then the dimensions i and j_L are swapped, which also changes the order of elements in memory.

5 Directing Data Layout Transformation

Intuitively, the space of all possible layouts that can be derived by applying the data layout transformation primitives arbitrarily on a multidimensional data structure can be very large. However, by leveraging properties from both the SPMD programming model, common on massively parallel systems, and the class of applications we are targeting, we demonstrate a generalizable data layout methodology for this application/target pair, based on an analytical model of the memory hierarchy and static analysis of the program. Finally, a data flow analysis is designed to help deduce data layouts for subscripted pointer accesses in the program.

5.1 Benchmarking and Modeling Memory System Characteristics

For massively parallel architectures such as the GPU, the number of concurrent memory requests from all the processors can be large, especially for codes with large datasets.

In such systems, DRAM controllers spread concurrent requests through the interconnect into different memory channels and banks, mostly by hashing address bits. Moreover, on some systems such as the NVIDIA G80 and GT200 GPUs, memory requests are vectorized (or coalesced, in CUDA terms) based on the least significant bits of their addresses if these requests are from a subset of threads that are executed in SIMD fashion (i.e. CUDA warps) by the underlying hardware.

To better understand how memory interleaving works, it is necessary to benchmark the underlying memory hierarchy to model the achieved memory bandwidth as a function of the distribution of memory addresses of concurrent requests. As an example, we derive an analytical model for an NVIDIA Tesla GPU, and use the execution model of that GPU to analyze the expected program execution flow and concurrent requests likely to be generated. Other devices and programming models could be evaluated independently with a similar approach. Previous work [31] benchmarked the GPU to explore memory latency as a function of access strides in a single-thread setting. However, since the class of applications we are targeting is mostly bandwidth-limited, we must determine how the effective *bandwidth* varies given access patterns across *all* concurrent requests. First, each memory controller will have some pattern of generating DRAM burst transactions based on requests. The memory controller could be only capable of combining requests from one core, or could potentially combine requests from different cores into one transaction. In our example, the GPU memory controller implements the former, with the CUDA programming manual [23] defining the global memory coalescing rule, which specifies how transactions are generated as a function of the simultaneous requests from the vector lanes of one streaming multiprocessor (SM).

Next, we must define our model on which bits in a memory address steer interleaving among memory channels, DRAM banks, or other parallel distribution structures built into the architecture to increase the number of concurrently satisfiable requests. We can determine these *steering bits* by observing the behavior of a microbenchmark generating concurrent requests with a fixed stride pattern and the resulting achieved bandwidth. Figure 3 illustrates this behavior for an NVidia Tesla GPU. The microbenchmark is similar to pointer-chasing in lmbench [19]: each thread repeats the statement `x = A[x]` for a large number of iterations, with the array `A` initialized with `A[i] = i` and each thread initialized with `x = blockIdx.x *` *`Stride`*. There is only one thread per thread block to ensure that each request results in one memory transaction.

By examining the spikes of poor bandwidth in Fig. 3, we can see a couple of features of the underlying system. First, each successive power-of-two stride essentially generates a concurrent set of requests with a fixed bit pattern in an increasingly large number of lower address bits. Continued performance degradation as the stride doubles indicates that the bit that was variant in the previous power of two but not in the current one is relevant to the parallel distribution of requests. Figure 3 shows that

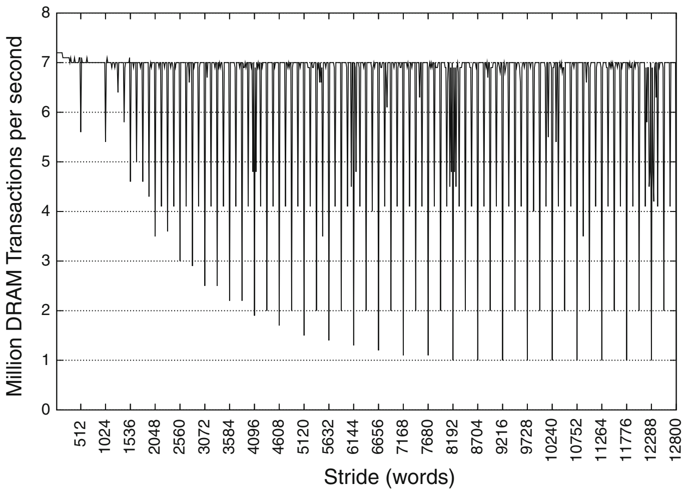

Fig. 3 Effective memory bandwidth vs. strides in words between requests from many single-threaded blocks on an Nvidia Tesla GPU. Bandwidth is in millions of transactions per second, and strides are in increments of 64 words

strides of 512, 1024, 2048, 4096 and 8192 words achieve successively lower effective bandwidths. Although more detailed microbenchmarks suggest that the interleaving is sophisticated enough that many of the higher bits may contribute to steering to some degree, the most critical bits are those at or below bit position 13. Moreover, the worst observed bandwidth occurs on strides with a multiple of 512 words (2K bytes), indicating that the 11 lowest bits have the most direct impact on the achieved MLP. For instance, note that strides of 8192+x*512 words are equally poor in performance as 8192 strides. Through further detailed microbenchmarking, we have confirmed that all bit positions in the range [13:6] are essential to spreading accesses to different memory channels and banks. Therefore, for the purposes of data layout of arrays of word-sized elements, we would consider the lower twelve bits of a flattened index expression to be relevant (equivalent to address bits [13:2]), and the bits in positions [10:6] the most important to vary across burst requests and indeed sufficient to distribute accesses across all memory system elements. From the coalescing rules [23], bits [5:2] are inferred to be offsets into a DRAM burst. Within a burst, a good layout transformation must maximize the number of useful words in that burst.

5.2 Data Transformation for Structured Grid Codes on a Two-level SPMD Programming Model

In current GPU architectures, each thread can only execute one memory operation at a time. Concurrent requests are therefore generated from different threads executing concurrently on the parallel hardware. Intuitively, index expressions dependent on thread and thread block identifiers should have significant variations in their values, and therefore variations in the bits representing the resulting address. To maximize bandwidth utilization, the intuitive goal of data layout transformation is to ensure that the address bits dependent on thread and thread block identifiers are the same as those bits used in the memory system to distribute concurrent requests among parallel memory system elements, and that the transformed access expressions adhere to the coalescing rules for full utilization of DRAM bursts.

Let us first consider the CUDA-like pseudo code in Listing 1 that is a simplified version of the 2D lattice-Boltzmann method (LBM):

Listing 1 Running Example

```
enum {N=0, E, W, S};

// Declare A0 and Anext as 2D variable—
// length arrays of 4—element structures

__global__ void
example(int ny, int nx, float A0[ny][nx][4],
float Anext[ny][nx][4])
{

int i = threadIdx.x+1, j = blockIdx.x+1;

// Access in FORTRAN—like form

float x_velo = A0[j][i][E] − A0[j][i][W];
float y_velo = A0[j][i][N] − A0[j][i][S];

Anext[j][i−1][E] = x_velo;
Anext[j][i+1][W] =−x_velo;
Anext[j−1][i][N] = y_velo;
Anext[j+1][i][S] =−y_velo;
}
```

In this code we have a 2D Array-of-Structures (AoS) layout. The code performs operations on the input cell owned by the thread, using the results to update specific fields of its neighbors in the output. Note that the leftmost dimension of every index expression is some constant value plus `blockIdx.x`, the second dimension is always some constant plus `threadIdx.x`, and the last dimension is a fixed offset denoting a structure field. The Array-of-Structures (AoS) layout is good for CPUs or cache-based architectures in general because of better spatial locality among structure members, but for GPUs this stops the memory vectorization hardware (or memory coalescing hardware in CUDA terms) from fully utilizing DRAM bursts when concurrent threads each requests a certain field of its own cell. The coalescing rules effectively state that the index of the lowest dimension must be dependent on `threadIdx` for good coalescing. This issue can be easily resolved by permuting the data layout, perhaps by exchanging the second and last dimensions, leading to addresses that satisfy the coalescing rule.

However, a good layout in terms of maximal MLP should also make concurrent memory accesses from different warps have distinct bits at the steering bits. Intuitively, we should not only make a vectorizable access pattern, but also assign bits of thread and thread block identifiers *most likely* to be distinct among active threads to those steering bits. Identifying which bits will be distinct among concurrent accesses requires analysis dependent on the execution model of the architecture. A good data layout would take these *busy bits* from the index of each dimension and map them into the steering bits of the memory system. A more formal definition and automated solution is presented in the remainder of this section.

5.2.1 Characterizing Thread Indices in Two-level SPMD Programming Models

In the two-level threading (thread/block) models employed by OpenCL and CUDA, some properties regarding thread indices can be observed:

– Computational grids consist of fixed-size thread blocks issued as a unit to the processors (i.e. SMs in CUDA terms), and executed asynchronously across them. As for thread IDs, asynchronous execution means that any thread with a legitimate thread ID within a block can be the issuer of a memory request.
– The total number of blocks in the computational grid can be very large, outnumbering the number of processors in the system, so the runtime issues a subset of these blocks to the processors. In other words, at any instant there is only a subset of X blocks being executed so the number of distinct block IDs usually is only a fraction of total number of blocks in a computational grid. With some simplifying assumptions about the regularity of block scheduling, the index range of blocks executing at a time can be roughly modeled as some oldest, still-executing block to some youngest executing block with an index of X plus the index of the oldest block minus one.

We can then characterize thread and block IDs in terms of distinct least significant bits across their concurrent instances:

– The number of distinct least significant bits across concurrent block IDs is about $\log_2$(maximum capacity of active blocks in the system)
– The number of distinct least significant bits across concurrent thread IDs is about $\log_2$(block size)

For CUDA, the maximum capacity for active blocks in the system can be determined statically from the compiled code's resource usage and the device parameters [23].

For our running example, assume there are 32 active thread blocks, each with 128 constituent threads, which means 5 LSBs of a thread block index and 7 LSBs of a thread index will be busy. In this case, one good layout for array `A0` could be created by strip-mining the Y and X dimensions by 32 and 128 respectively and shifting the resulting sub-dimensions into the steering and coalescing bit positions.

In terms of dimension vectors and flattening functions, the dimension vector of `A0` is $\mathbf{D} : (\lceil \texttt{ny}/2^5 \rceil, \lceil \texttt{nx}/2^7 \rceil, 4, 2^5, 2^7)$, where `nx` and `ny` are from C99 VLA declaration of `A0`; the FF of `A0` is $FF(\mathbf{I}, \mathbf{D}) : I_{2[:5]} D_3 D_2 D_1 D_0 + I_{1[:7]} D_2 D_1 D_0 + I_0 D_1 D_0 + I_{2[4:0]} D_0 + I_{1[6:0]}$, where $\mathbf{I}$ is the index vector of the array subscripts, e.g. for `A0[j][i][0]`, $\mathbf{I} : (I_2 = j, I_1 = i, I_0 = 0)$.

5.2.2 *Automated Discovery of Ideal Data Layouts*

To automate the process of selecting and shifting bits to best fit the memory system, we begin with a high-level algorithmic description of the procedure:

1. Convert all grid-accessing expressions into affine forms of thread and thread block indices and surrounding sequential loop indices. For structured grid codes that use FORTRAN-like array subscripts, array accessing expressions can be usually converted to this form. In principle, if there are non-affine terms in an expression, we could still approximate it by introducing auxiliary affine terms, as suggested by Girbal et al. [8].
2. For a given grid, if all expressions accessing the grid share the same coefficients for all columns except the constant column, then this grid is eligible for layout transformation. We call the grid *eligible*, and define a matrix consisting of the coefficients of the affine form of its accessing expressions, except the constant column, as the grid's *common access pattern*. For structured grid codes which access nearest neighbors, the access expressions of the same grid usually have the same coefficients except for the last column. E.g. `[x+1][y]` and `[x-1][y-1]` are considered of the same common access pattern $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{pmatrix} \texttt{x} \\ \texttt{y} \end{pmatrix}$.
3. For each eligible grid, derive the desired data layout from its common access pattern:
   (a) Calculate the number of busy bits of each referred thread and block index from the occupancy and thread block configuration.
   (b) For each dimension, compute the collective busy bits represented by the corresponding row in the common access pattern. Since a row in the common access pattern represents some linear combination of thread and block indices, the collective busy bits are the union of these busy bits, while some of them are possibly shifted by log_2 of their coefficients.
   (c) Assign the least significant N bits of the fastest changing dimension index to the bit position that is used for memory coalescing, where N is the number of address bits that determine memory vectorization according to the hardware specification.
   (d) Greedily assign other collective busy bits from all dimensions to the steering bits by strip-mining power-of-two-sized tiles and permuting these tiles to the desired bit positions until all steering bits are occupied or there are no busy bits left from any dimension.
   (e) Assign all remaining bits to the higher dimensions.
   (f) Generate flattening functions and dimension vectors according to the above assignment and the C99 VLA declaration for the grid.

4. Perform dataflow analysis to derive the flattening function associated with each array accessing expression.
5. Output the transformed code with inline-expanded flattening function at grid accessing expressions.

For our running example, some of the access functions of `A0` and `Anext` are: $(blockIdx.x + 1, threadIdx.x + 1, E)$, $(blockIdx.x + 1, threadIdx.x + 1, W)$, $(blockIdx.x, threadIdx.x + 1, N)$, and $(blockIdx.x + 2, threadIdx.x + 1, S)$. In the affine form of access functions similar to the notation used by Girbal et al. [8], they would look like: $\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & \texttt{E} \end{bmatrix} \begin{pmatrix} \texttt{blockIdx.x} \\ \texttt{threadIdx.x} \\ 1 \end{pmatrix}$, $\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & \texttt{W} \end{bmatrix} \begin{pmatrix} \texttt{blockIdx.x} \\ \texttt{threadIdx.x} \\ 1 \end{pmatrix}$, $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & \texttt{N} \end{bmatrix} \begin{pmatrix} \texttt{blockIdx.x} \\ \texttt{threadIdx.x} \\ 1 \end{pmatrix}$, and $\begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 1 \\ 0 & 0 & \texttt{S} \end{bmatrix} \begin{pmatrix} \texttt{blockIdx.x} \\ \texttt{threadIdx.x} \\ 1 \end{pmatrix}$, respectively. The affine form of the access functions of `Anext` and `A0` only differ in the last column, and thus both arrays are eligible for data layout transformation, with their common access pattern being $\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}$. The common access pattern clearly links the dimension with individual thread and thread block indices, which are used for deciding the actual layout based on their busy bits.

Continuing with our previous example, we will assume the number of active thread blocks is 32, and the number of threads in the thread block is 128. This means that the 5 least significant bits of `blockIdx.x` are busy, and the all seven meaningful bits of `threadIdx.x` are busy. The second dimension index, corresponding to `threa-dIdx.x`, takes the lowest dimension place in the transformed layout for coalescing (4 least significant bits) and the first three steering bits in a word address. The highest dimension is split into two dimensions, with the 5 least significant bits accessing the new lower dimension and the remaining bits accessing the higher dimension. The newly created lower dimension is transposed to take the second lowest dimension of the new layout. The remaining dimensions are left as they are, resulting in the layout shown before.

5.3 Propagating Layout Information as Extended Types with Pointers

After each solver iteration, iterative PDE solver implementations in C or C-like languages usually swap pointers to the input and output grid before starting the next iteration, i.e. the output of the current iteration becomes the input of the next iteration. Hence, correct propagation of layout and dimension information through pointer assignments is essential for these solver implementations.

In other words, after deciding the layout of a specific grid, we need to analyze the source code to figure out the set of grid access expressions, in the form of subscripted pointer dereferences, that need to be updated to use the transformed flattening function instead. We address this issue by treating layouts as extended types and solve the dataflow equation to analyze the layout for array accessing expressions.

Types in programming languages specify the information necessary for the code to interpret and operate on instances of that type. The layout of an array is an implicit part of an array's type, typically defined by the language. To transform the layout of a particular array, excluding other arrays, we must essentially change that array's type, and propagate that change in type information through the program to ensure that all parts of the program accessing that array do so correctly. This propagation could be performed at runtime by extending the array type in the compiler to augment the grid with a function pointer to the flattening function, set when the array is allocated. However, current GPU programming models do not allow indirect calls, so we elect to perform the propagation of the type change instigated by the compiler in the compiler itself.

Therefore, we present algorithms for propagating the implicit layout type information statically through a program, identifying the pointer references that access the objects with extended types. The proposed usage scenario is that the user specifies through annotation the grid on which the compiler should perform automatic layout transformation, without specifying the actual layout, and the compiler decides the layout that works best on the given grid for a given architecture, and propagates this layout information through this analysis.

Our approach involves a source-to-source compiler that transforms the flattening function of expressions accessing grids annotated with dimension vectors, effectively deriving layout-transformed arrays, and finally emits CUDA C code that can be further compiled by the NVCC compiler with inline-expanded flattening functions on dynamically-allocated one-dimensional arrays.

We formulate this analysis as a monotonic dataflow analysis. In this framework, a dataflow analysis is represented as a meet-semilattice and a set of transfer functions. For this problem, the semilattice is $(\Psi, \wedge)$, where each element in the semilattice is a function: $\Psi : P \rightarrow \mathbb{L} \cup \{UT, \perp\}$. P is the set of pointer variables in the program, UT stands for *untransformed* and $\perp$ means *incompatible* respectively. $\mathbb{L}$ is a set containing the definitions of new data layouts, each fully defined by a dimension $n \in \mathbb{N}$, a dimension vector $\mathbb{N}^n$ and a flattening function $\mathbb{N}^n \rightarrow \mathbb{N}$. When this function maps a pointer to a new layout, it is asserting that every data structure the pointer may refer to shares the specified layout. An untransformed pointer indicates that the data structure it points to uses *RML* as its flattening function; an incompatible pointer, however, indicates that this pointer may point to at least two data structures with incompatible flattening functions. Two flattening functions FF_1 and FF_2 are compatible (expressed as $FF_1 == FF_2$) if and only if for all legitimate dimension vectors $\mathbf{D}$ and index vectors $\mathbf{I}$, $FF_1(\mathbf{D}, \mathbf{I}) = FF_2(\mathbf{D}, \mathbf{I})$. That is, the FF for a `float` array can be compatible with the *RML* for a `long` array as long as their element sizes are the same. This allows transforming the layout of some structured-grid code, in which non-float typed elements are accessed through type-casted grid base pointer.

The set of transfer functions $f : \Psi \rightarrow \Psi$ is created from the operation types in the flow graph as shown in Table 1. The meet operation of two functions $m, n \in \Psi$ is defined in Table 2. In the table, the binary relationship $==$ for two tuples $\{l1 = (n_1, D_1 \in \mathbb{N}_{expr}{}^{n_1}, FF_1), l2 = (n_2, D_2 \in \mathbb{N}_{expr}{}^{n_2}, FF_2)\} \in \mathbb{L}$ exists if and only if $n_1 = n_2$ and $D_1 = D_2$ and $FF_1 == FF_2$. In a word, each statement, according to its operation type, may change the layout bound to a pointer through assignment.

Table 1 Transfer functions

Operation type	Transfer function $f(\mu)$ in the form of $f(\mu) = \nu$ with $\nu(w) = \mu(w) \forall (w \neq \texttt{p1})$ and $\nu(\texttt{p1}) = \ldots$, where $w \in P$; $\mu, \nu \in \Psi$
No definition involving any pointer variables	$\nu(\texttt{p1}) = \mu(\texttt{p1})$ (Identity function)
`p1 = p2;` `p1` and `p2` are pointers	$\nu(\texttt{p1}) = \mu(\texttt{p2})$
`p1 = p2 + t;` `p1` and `p2` are pointers and `t` is of integer type	$\nu(\texttt{p1}) = \perp$ if $\mu(\texttt{p1}) \neq UT$ else UT
Declaring a pointer `p`	$\nu(\texttt{p1}) = UT$
Declaring a pointer `p` to an n-dimensional grid `G` with a dimension vector `DV`	$\nu(\texttt{p1}) = (n, \texttt{DV}, RML)$
Apply layout transformation `lt` to the data structured pointed by `p1`	$\nu(\texttt{p1}) = \texttt{lt}(\mu(\texttt{p1}))$ where `lt` is a layout transformation

Table 2 Meet function $\wedge$

	`l1`	UT	$\perp$
`l2`	if `l1` == `l2` then `l1` else $\perp$	$\perp$	$\perp$
UT	$\perp$	UT	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$

Transformed and untransformed layouts, as well as dimension vectors of grids, are thus propagated.

The meet function $\wedge$ deals with the join of control flows. Since most programming models for the GPU do not allow indirect function calls in general, for each grid access expression only one flattening function is allowed to bind with that expression. The meet function basically aborts data layout transformation for a particular grid if there are multiple incompatible flattening functions that need to be bound with any expression that accesses the grid (i.e. the binary relation == does not hold for these functions). This restriction can surely be slightly relaxed using versioning, but this is left for future work.

6 Experimental Results

Three CUDA benchmarks, namely CFD, Heat [5], and LBM [24], were used to explore the significance of memory-level parallelism for memory-bound structured grid applications and the validity of the data layout transformation heuristic presented in the paper. CFD is an implementation of the red-black Gauss-Seidel method for a 3D Navier-Stokes solver, Heat is a 3D heat equation solver using the Jacobi method, and LBM is an implementation of the SPEC2006CPU [30] Lattice-Boltzmann method. The first two benchmarks represent the two major point methods for solving PDEs using the finite difference method. LBM is an alternative CFD approach using a particle-based method instead of discretizing the PDE. For each benchmark, the performance of different layouts is presented in terms of the normalized execution time over several

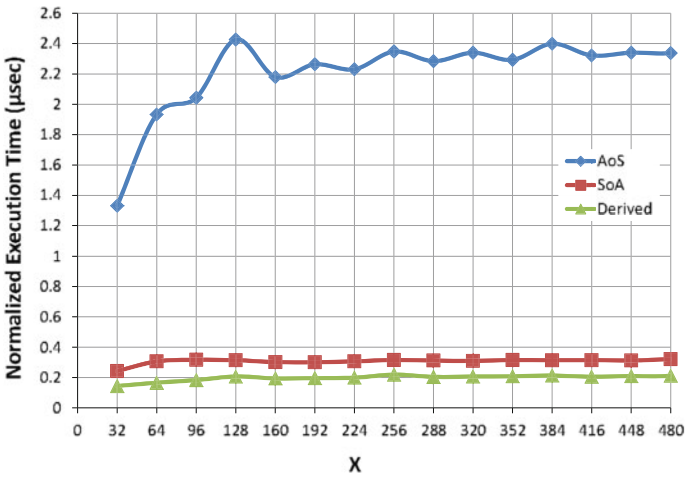

Fig. 4 LBM: varying X

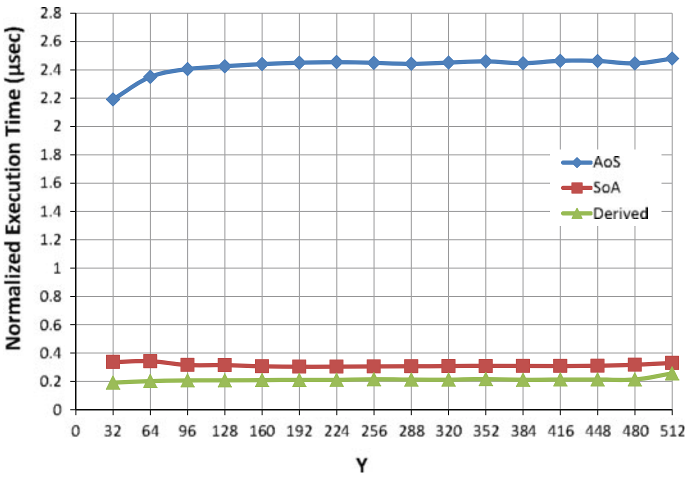

Fig. 5 LBM: varying Y

ranges of grid sizes, changing the size of one dimension at a time. The experiments were run on an Nvidia Tesla T10 GPU with 4GB of memory.

We first manually convert each of the benchmarks into a layout-neutral form and apply our automated layout transformation methodology on the main grids on which it operates. Because our compiler infrastructure does not yet support variable-length array syntax, we use annotations to communicate that information to the compiler. After automatic transformation, the nearby regions of the space of potential layout transformations where the solution was found are manually searched for the best candidate.

The results show the criticality of a data layout for maximizing bandwidth utilization by both vectorizing memory accesses into bursts, and parallelizing them across

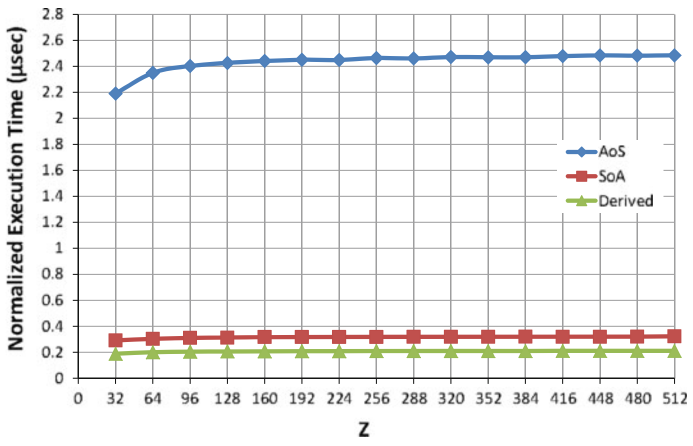

Fig. 6 LBM: varying Z

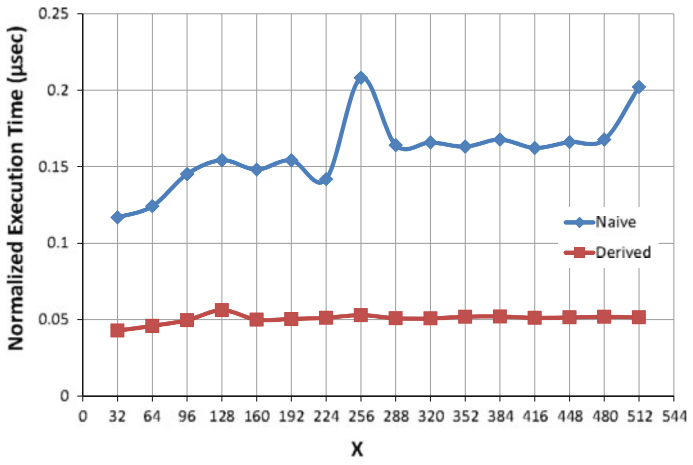

Fig. 7 Heat: varying X

interleaved memory channels and banks. The relative performance of a layout depends on its divergence from the optimal layout in both of these two criteria.

For the LBM benchmark, Figs. 4–6 contrast the performance of the layout derived from the transformation heuristic to the array-of-structures (AoS) and structure-of-arrays (SoA) layouts. On average, switching from the AoS layout to the SoA layout improves the performance by 7.2X, mainly due to improved burst-level parallelism from better memory coalescing. However, the layout which maps busy bits to steering bits more prudently, thereby achieving higher memory-level parallelism, further improves the performance by 1.52X. Moreover, such a layout is more persistent to grid size variations.

Figures 7–9 show the merits of using an MLP-aware layout for the Heat benchmark over a layout oblivious to it. While both layouts result in fairly coalesced memory

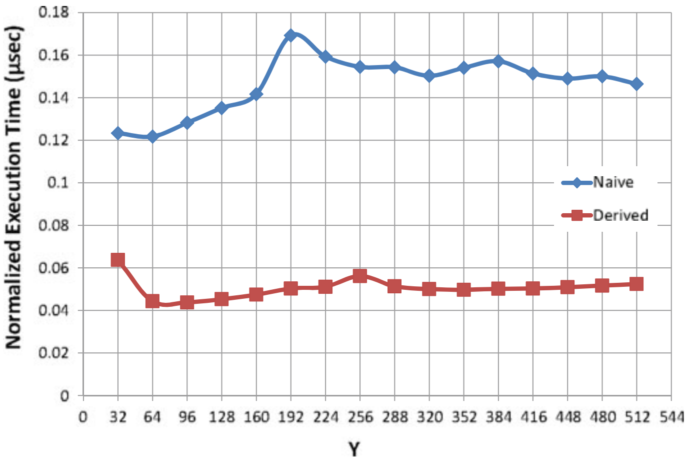

Fig. 8 Heat: varying Y

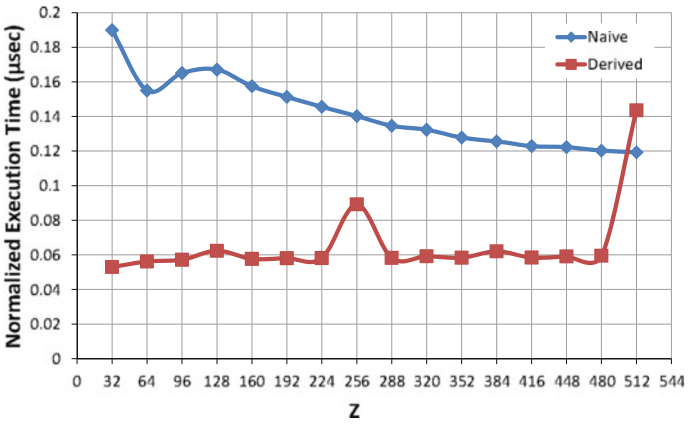

Fig. 9 Heat: varying Z

access patterns, the layout derived from the transformation heuristic is 2.74X faster on average.

Figures 10–12 compare the performance of the layout of the CFD benchmark derived from the transformation heuristic to the default row-major layout (RML) defined by the programming language. Effective tiling for the memory interleaving hardware, which also results in marginally better memory coalescing, improves the performance of the derived layout by 1.16X on average over RML.

Our experiments show that even with extra overhead computing memory addresses, the transformed benchmarks still gain performance by improving the efficiency of accessing memory. This highlights both the bandwidth-boundedness of the benchmarks themselves, and the validity of trading extra address calculation instructions for better bandwidth utilization in bandwidth-bound applications.

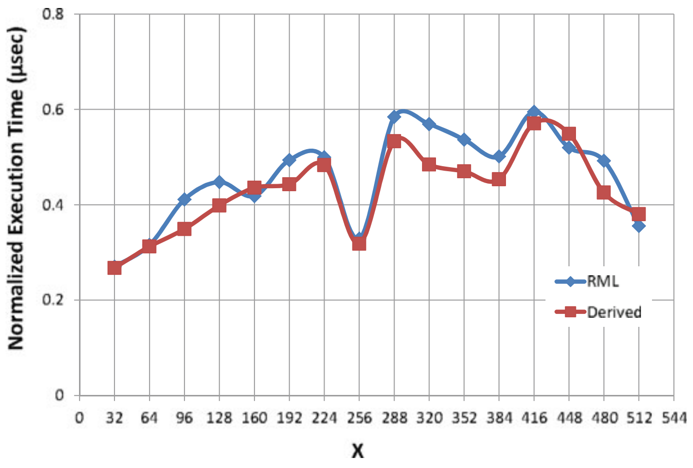

Fig. 10 CFD: varying X

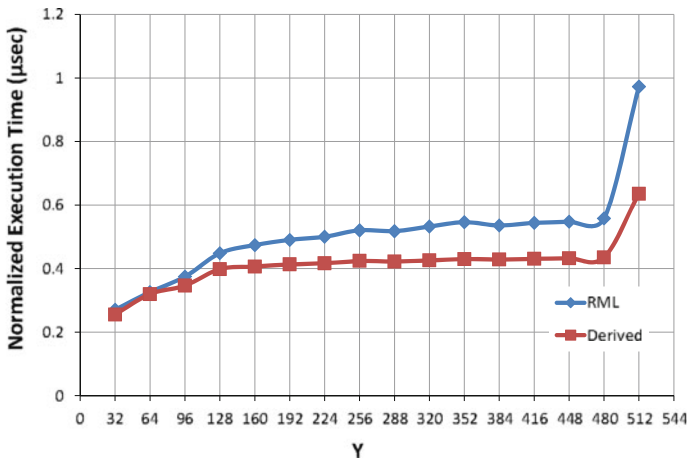

Fig. 11 CFD: varying Y

7 Conclusion and Future Work

We presented a formulation and language extension that enable automatic data layout transformation for structured grid codes in CUDA. We also benchmarked an Nvidia Tesla GPU to reveal its DRAM banking and interleaving scheme. Based on the microbenchmark results, we developed a layout transformation methodology that can significantly speed up various structured-grid codes by distributing concurrent memory requests evenly to DRAM channels and banks.

Our methodology does not preclude opportunities of applying other transformations that aim at improving reuse. Future work investigating holistic data layout transformations addressing temporal locality, spatial locality, and MLP will be

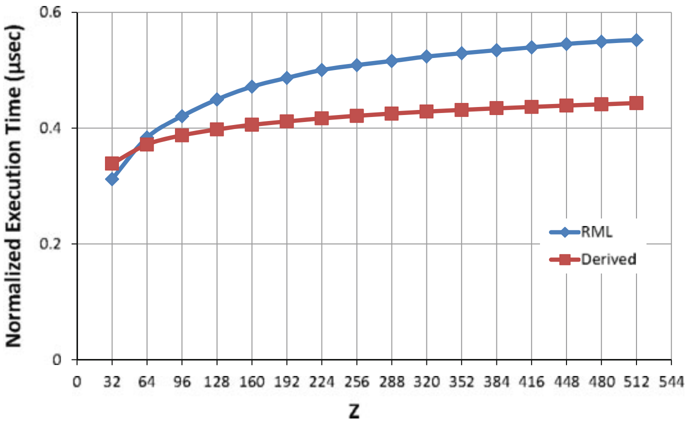

Fig. 12 CFD: varying Z

paramount to achieving the highest levels of performance for important, bandwidth-bound structured grid applications.

Acknowledgments This work was funded by the Universal Parallel Computing Research Center at the University of Illinois at Urbana-Champaign. The Center is sponsored by Intel Corporation and Microsoft Corporation. This work utilized the AC cluster [14] operated by the Innovative Systems Laboratory (ISL) at the National Center for Supercomputing Applications (NCSA) at the University of Illinois. The cluster was funded by NSF SCI 05-25308 and CNS 05-51665 grants along with generous donations of hardware from NVIDIA, Nallatech, and AMD. We would like to thank Chris Rodrigues, Nady Obeid, and anonymous reviewers for their comments.

References

1. Anderson, J.M., Amarasinghe, S.P., Lam, M.S.: Data and computation transformations for multiprocessors. SIGPLAN Not. **30**(8), 166–178 (1995)
2. Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., Yelick, K.A.: The landscape of parallel computing research: a view from berkeley. Technical report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006
3. Bakhoda, A., Yuan, G.L., Fung, W.W.L., Wong, H., Aamodt, T.M.: Analyzing cuda workloads using a detailed gpu simulator. In: ISPASS, pp. 163–174. IEEE (2009)
4. Baskaran, M.M., Bondhugula, U., Krishnamoorthy, S., Ramanujam J., Rountev, A., Sadayappan, P.: A compiler framework for optimization of affine loop nests for gpgpus. In: ICS ’08: Proceedings of the 22nd annual international conference on Supercomputing, pp. 225–234. ACM, New York, NY, USA (2008)
5. Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., Yelick, K.: Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: SC08: Proceedings of the 2008 Conference on Supercomputing, pp. 1–12. Piscataway, NJ, USA (2008)
6. Demmel, J.W.: Applied Numerical Linear Algebra. Society for Industrial and Applied Mathematics, Philadelphia, PA (1997)
7. Ferziger, J.H., Peric, M.: Computational Methods for Fluid Dynamics. Springer, Berlin (1999)
8. Girbal, S., Vasilache, N., Bastoul, C., Cohen, A., Parello, D., Sigler, M., Temam, O.: Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. Int. J. Parallel Prog. **34**(3), 261–317 (2006)

9. Gundolf, C.D., Douglas, C.C., Haase, G., Hu, J., Kowarschik, M., Weiss, C.: Portable memory hierarchy techniques for PDE solvers, part II. SIAM News **33**, 8–9 (2000)
10. Ipek, E., Mutlu, O., Martínez, J.F., Caruana, R.: Self-optimizing memory controllers: A reinforcement learning approach. Comp. Arch. News **36**(3), 39–50 (2008)
11. Jang, B., Mistry, P., Schaa, D., Dominguez, R., Kaeli, D.: Data transformations enabling loop vectorization on multithreaded data parallel architectures. In: PPoPP '10: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 353–354. ACM, New York, NY, USA (2010)
12. Ju Y.-L., Dietz, H.G.: Reduction of cache coherence overhead by compiler data layout and loop transformation. In: Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing, pp. 344–358. Springer, London, UK (1992)
13. Kennedy, K., Kremer, U.: Automatic data layout for high performance fortran. In: Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM), pp. 76. ACM, New York, NY, USA (1995)
14. Kindratenko, V., Enos, J., Shi, G.: Gpu clusters for high-performance computing. In: Proceedings of the Workshop on Parallel Programming on Accelerator Clusters. Jan 2009
15. Kwon, Y.-S., Koo, B.-T., Eum, N.-W.: Partial conflict-relieving programmable address shuffler for parallel memories in multi-core processor. In: ASP-DAC '09: Proceedings of the 2009 Asia and South Pacific Design Automation Conference, pp. 329–334. IEEE Press, Piscataway, NJ, USA (2009)
16. Lu, Q., Alias, C., Bondhugula, U., Henretty, T., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P., Chen, Y., Lin, H., Ngai, T.-f.: Data layout transformation for enhancing data locality on nuca chip multiprocessors. In: Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques, pp. 348–357 (2009)
17. Mace, M.E.: Memory Storage Patterns in Parallel Processing. Kluwer, Boston (1987)
18. Mahapatra, N.R., Venkatrao, B.: The processor-memory bottleneck: problems and solutions. Crossroads **5**(3es), 2 (1999)
19. McVoy, L., Staelin, C.: lmbench: portable tools for performance analysis. In: Proceedings of the 1996 USENIX Annual Technical Conference, pp. 23–23 (1996)
20. Morton, K.W., Mayers, D.F.: Numerical Solution of Partial Differential Equations: An Introduction. Cambridge University Press, New York, NY (2005)
21. Moscibroda, T., Mutlu, O.: Distributed order scheduling and its application to multi-core DRAM controllers. In: Proceedings of the 27th Symposium on Principles of Distributed Computing, pp. 365–374 (2008)
22. Mutlu, O., Moscibroda, T.: Parallelism-aware batch scheduling: enhancing both performance and fairness of shared DRAM systems. Comput. Arch. News **36**(3), 63–74 (2008)
23. nVIDIA: nvidia cuda programming guide 2.0 (2008)
24. Pohl, T., Kowarschik, M., Wilke, J., Iglberger, K., Rüde, U.: Optimization and profiling of the cache performance of parallel lattice boltzmann codes. Parallel Process. Lett. **13**(4), 549–560 (2003)
25. Qian, Y.H., D'Humieres, D., Lallemand, P.: Lattice BGK models for Navier-Stokes equation. Europhys. Lett. **17**(6), 479–484 (1992)
26. Rivera, G., Tseng, C.-W.: Tiling optimizations for 3D scientific computations. In: SC00: Proceedings of the 2000 conference on Supercomputing, p. 32 (2000)
27. Ryoo, S., Rodrigues, C.I., Baghsorkhi, S.S., Stone, S.S., Kirk, D.B., Hwu, W.-m.W.: Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In: Proceedings of the 13th Symposium on Principles and Practice of Parallel Programming, pp. 73–82 (2008)
28. Sellappa, S, Chatterjee, S.: Cache-Efficient multigrid algorithms. Int. J. High Perform. Comput. Appl. **18**(1), 115–133 (2004)
29. Shao, J., Davis, B.T.: A burst scheduling access reordering mechanism. In: Proceedings of the 13th International Symposium on High Performance Computer Architecture, pp. 285–294 (2007)
30. Spradling, C.D.: Spec cpu2006 benchmark tools. Comput. Arch. News **35**(1), 130–134 (2007)
31. Volkov, V., Demmel, J.W.: Benchmarking gpus to tune dense linear algebra. In: SC08: Proceedings of the 2008 Conference on Supercomputing, pp. 1–11 (2008)
32. Zhao, Y.: Lattice Boltzmann based PDE solver on the GPU. Vis. Comput. **24**(5), 323–333 (2008)