Data Layout Transformation for Structured-Grid Codes on GPU

I-Jui Sung, Wen-Mei Hwu
University of Illinois at Urbana-Champaign
{sung10, w-hwu}@uiuc.edu

***Abstract*—We present data layout transformation as an effective performance optimization for memory-bound structured-grid applications for GPUs. Structured grid applications are a class of applications that compute grid cell values on a regular 2D, 3D or higher dimensional regular grid. Each output point is computed as a function of itself and its nearest neighbors. Stencil code is an instance of this application class. Examples of structured grid applications include fluid dynamics and heat distribution that solve partial differential equations with an iterative solver on a dense multidimensional array.**

Using the information available through variable-length array syntax, standardized in C99 and other modern languages, we have enabled automatic data layout transformations for structured grid codes with dynamic array sizes. We first present a formulation that enables automatic data layout transformations for structured grid code in CUDA. We then model the DRAM banking and interleaving scheme of the GTX280 GPU through microbenchmarking. We developed a layout transformation methodology that guides layout transformations to statically choose a good layout given a model of the memory system. The transformation which distributes concurrent memory requests evenly to DRAM channels and banks provides substantial speedup for structured grid application by improving their memory-level parallelism.

I. Introduction

Structured grid applications [1] 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 patterns more general than a fixed stencil. Examples of structured grid applications include fluid dynamics and heat distribution that solve partial differential equations (PDEs) with an iterative solver on dense multidimensional arrays. When parallelizing such applications, the most common approach is to spatially partition the grid cell computations into fixed sized chunks, usually as planes or cuboids, and assign those chunks to workers e.g. threads, MPI ranks, or through OpenMP parallel for loops.

However, the underlying memory hierarchy may not interact in the most efficient way with a given decomposition of the problem. Data accesses may not fully exploit parallelism among memory controllers, interleaved DRAM banks, elements within a DRAM burst (pipelined access of a subset of DRAM columns in a row). Furthermore, the full details of the memory hierarchy are often too obscure or complex for a typical application programmer to make the best use of them. Even for exceptional cases where the programmer does know how to transform the data layout to fit the memory system,

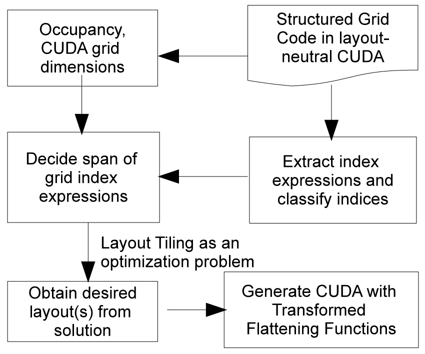
Occupancy, CUDA grid dimensions
Structured Grid Code in layout-neutral CUDA
Decide span of grid index expressions
Extract index expressions and classify indices
Layout Tiling as an optimization problem
Obtain desired layout(s) from solution
Generate CUDA with Transformed Flattening Functions

Fig. 1: Data Layout Transforms for Structured Grid Codes

performing the transformation manually is very tedious, results in less readable code, and must be transformed again every time a new platform is targeted.

Currently, programming languages such as C and FORTRAN rigidly define the layout of natural arrays, and allow usages that rely on that natural layout, such as casting to a linear array or deterministically aliasing “out-of-bounds” accesses to real elements. 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 layout using a new language data type or C++-style classes both complicate the language and may contain undesirable overheads for accessing the most performance-critical data structure of the application. The data layout optimization proposed in this paper addresses these problems by defining a layout-neutral form of array allocation and access, and defining transformations that allow us to adapt the linear representation to best utilize the memory-level parallelism (MLP) of the underlying memory hierarchy.

Figure 1 depicts our procedure of data layout transformation, using a modern GPU as an example memory system. The input is a kernel in layout-neutral form, which can be considered 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. 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 transformed code with array access expressions converted to flattened array accesses using transformed layouts.

The rest of this paper explains our methodology and results in detail. Section II provides an overview of iterative PDE solvers. Section III discusses related work in data layout transformations. Section IV-B formulates logical and physical representations of arrays, and defines the data layout transformations we consider using this formulation. Section V discusses how we obtained a memory address interleaving scheme of the DRAM controller through micro-benchmarking, and derive an optimized layout from the program and execution model. Section VI presents our experiment results, followed by some concluding remarks in Section VII.

II. 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 [2] or Finite Volume Methods [3], while solutions based on Finite Element Methods [2] 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 [4]. For large problems, direct-solution methods are often not viable: practical approaches are almost exclusvely iterative-convergence methods.

Iterative techniques like Jacobi method and Gauss-Seidel (including those with Successive Overrelaxation) are often used as important building blocks for more advanced solvers like multigrid [5]. Those methods are both instances of stencil codes, whose stencils can be expressed as a weighted sum of the cell and 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 at the same sweep; two separate, serialized sweeps over the red and black cells performs one whole iteration update.

The lattice-Boltzmann method (LBM) [6], a particle-based method that was mainly used in computational fluid dynamics problems, was recently extended as a general PDE solver [7]. The LBM is also an iterative method applied to structured grids. The cell update rules for the 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 [8]. The stream stage accesses the nearest neighbors of the current cell, while the collide stage's inputs are entirely local to the current cell.

III. 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 them 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 cache line size [9], [10].

Lu et al.'s recent work applies data layout transformation for cache locality in NUCA (non-uniform cache architecture) chip multiprocessors [11]. They employ similar data layout transformations, but their work targets localizing accesses to local L2 cache bank, rather than exploiting MLP in multi-cores connect to multi-channel memory controllers through interconnect, and only considers sequential loop indices to optimize cache locality.

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 use in this paper [12]. 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. [12].

All of the methods mentioned in this section thus far improve how efficiently data is used or reused in the on-chip cache of the system. However, these approaches are not always applicable or sufficient. For example, LBM within one timestep does not contain any data reuse [8], and even once reuse is exploited, some stencil codes may still be performance bound by off-chip bandwidth. Applications in such situations could potentially still gain significant performance improvement by using MLP-oriented optimizations.

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. However, for a massively parallel system, balancing DRAM traffic across controllers can be important. Datta et al. [12] take into consideration the affinity of DRAM controller and processor cores in NUMA architectures using an affinity-aware memory allocator. As far as we know, there is no software-based approach to balancing workloads for the multi-channel, interleaved DRAM controllers employed in modern parallel architectures, although according to our micro-benchmarks, 7X performance loss could occur in extreme cases.

For GPUs, we know of no previous work applying data layout transformation to structured-grid code other than simple array-of-structure to structure-of-array transformation [13], which exploits pipeline parallelism within DRAM bursts (i.e. coalescing). Our data layout transformation further exploits concurrency in multi-channel and interleaved DRAM bank organization.

Various works [14]–[17] have proposed hardware optimizations for DRAM controllers working toward uniform access latency and fairness, some parallelism aware [15], [16]. All of them focus on scheduling DRAM requests under certain workloads, therefore no attention is given to potentially less costly software approaches that transform the distribution of memory addresses requested. Also, those approaches only balance among memory banks under the same controller, while several controllers are are often present in modern systems.

IV. The Data Layout Transformation for Structured Grid C Code

In C, an array is a collection of values identified by integer indices with a language-defined memory layout. To enable layout transformation, we must first separate the use of indices from the linear layout. We first present a formalization of arrays, layouts, and layout transformation. Then, we describe how a compiler can recognize the information necessary to perform effective layout transformations efficiently, noting that the most reliable method is available only when the programmer specifies array accesses in FORTRAN-like multidimensional indexes. Variable-length array syntax, a recently standardized feature of the C language, enables that form of index expression even for arrays of all kinds, including those whose size is not statically known. Finally, a data flow analysis is designed to help deduce data layouts for subscripted pointer accesses.

A. Grids and Flattening Functions

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

An array element is identified by a vector of integers called an *index vector*. Without loss of generality, for the index vector $\vec{I}$ 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. T is the type of all elements in G.

Definition 2. *An injective function FF: $\mathbb{N}^n \to \mathbb{N}$ 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 that 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 many-to-one mapping, and thus f^{-1} is defined and $f^{-1}(f(\vec{I})) = \vec{I}$ for a valid index vector $\vec{I}$. With these restrictions, a flattening function uniquely defines a memory layout and vice-versa; we use these terms interchangeably in the remaining text.

B. Row-major layout and layout transformations

Theorem 1. *For an n-dimensional array with index vector $\vec{I}$ and dimension vector $\vec{D}$ with $0 \le I_i < D_i, \forall(0 \le i < n)$, function $RML(\vec{I}, \vec{D}) = \sum_{i=0}^{n-1} I_i \prod_{j=0}^{i-1} D_j$ is a flattening function that reduces $\vec{I}$ to an integer.*

Proof: Given the integer $i = RML(\vec{I}, \vec{D})$ for an element with index vector $\vec{I}$ of an n-dimensional array G with dimension vector $\vec{D}$, we can compute I_i by $(i/\prod_{j=0}^{i-1} D_j) \bmod D_i$. It follows that $0 \le I_i < D_i$. Since this applies to any legitimate array cell in G, it is clear that the function RML is injective, and is defined for all valid indices of G, so RML is a flattening function. ■

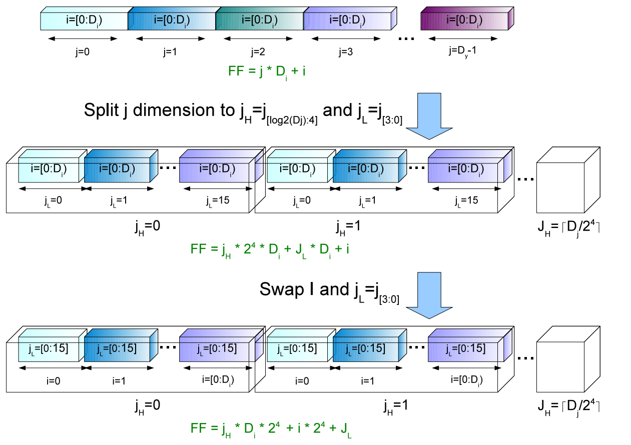

Fig. 2: An Example of Layout Transformation

C, and languages derived from it, define the layout of arrays as row-major order. For example, the RML for the array `a[3][4]` would be $I_0 + I_1 \times 4$. For an expression `a[j][i]`, RML computes $i + j * 4$. The inverse function $RML^{-1}(x)$ returns a vector $\vec{I} = \{x \bmod 4, x/4\}$.

We can define some transformations of RML with regards to an n-dimensional array G with index vector $\vec{I}$ and dimension vector $\vec{D}$:

Swap: Interchange index i and j ($i \ne j$). This transformation creates a new index vector $\vec{I'}$ from $\vec{I}$ and a new dimension vector $\vec{D'}$ from $\vec{D}$. $\vec{I'}$ and $\vec{D'}$ are then used as inputs to the transformed FF: $\sum_{i=0}^{n-1} I'_i \prod_{j=0}^{i-1} D'_j$ where:

$$\vec{I'} = \{I'_k, k \in \mathbb{N}, k < n\}; \quad I'_k = \begin{cases} I_k & \text{for } k \notin \{i, j\} \\ I_j & \text{for } k = i \\ I_i & \text{for } k = j \end{cases}$$

$$\vec{D'} = \{D'_k, k \in \mathbb{N}, k < n\}; \quad D'_k = \begin{cases} D_k & \text{for } k \notin \{i, j\} \\ D_j & \text{for } k = i \\ D_i & \text{for } k = j \end{cases}$$

Intuitively, this transformation swaps I_i and I_j as well as D_i and D_j. Considering multi-element structures as another array dimension, the common structure-of-arrays to array-of-structures transformation [13] is an instance of (repeated) application of the swap transform on neighboring dimensions, shifting the structure offset index to the highest position in the index vector. Repeated applications of swapping are required in general when it is desired to shift a dimension along some direction, so that the relative order of other dimensions is not changed.

Split: Split dimension i into T-sized tiles, $0 \le T < D_i$. This transformation creates a new index vector $\vec{I'}$ and a new dimension vector $\vec{D'}$, which are inputs to the transformed FF. $\vec{I'}$ and $\vec{D'}$ are created by dividing I_i into I_h, I_l and D_i into D_h, D_l, where $I_h = I_i \bmod T$, $I_l = \lfloor I_i/T \rfloor$ and $D_h = \lceil D_i/T \rceil$, $D_l = T$.

Swapping can be considered a "relabeling" of some indices and dimensions. Splitting is diving one index into two, with the lower index being a single-digit, base T number. This introduces "padding" into the original array dimension, bringing it to an even multiple of T. Since the new dimension is rounded up to the nearest multiple of T, it allows all valid indices in both portions of a split index to be uniquely mapped. It follows that these two operations are closed on FF. Practically, we only consider and implement split operations for $T = 2^k$, so the transformed FF can be expressed in fast bit operations rather than slow general integer modulo operations. Applying a split FF with power-of-two tile size on index vector $\vec{I}$ w.r.t. an array A is abbreviated as A$[I_{n-1}]$....$[I_{i(\log_2(D_i):k+1)}][I_{i(k:0)}]$...$[I_1][I_0]$. Figure 2 shows a layout tiling example that transforms an access to array $\texttt{A}[D_j][D_i]$ from `A[j][i]`, i.e. RML_A, to $\texttt{A}[\texttt{j}_{\log 2(\texttt{D}_\texttt{i}):4}][\texttt{i}][\texttt{j}_{3:0}]$. 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.

C. The Layout-Neutral Form and LN Function

Definition 3. *LN is a function* $\mathbb{S} \rightarrow \{\cup_{n\in\mathbb{N}} G_n \times {\mathbb{N}_{\text{expr}}}^n \times {\mathbb{N}_{\text{expr}}}^n\} \cup \{\epsilon\}$, *where* $\mathbb{S}$ *is the set of array-accessing expressions in a program P and the triple* $G_n \times {\mathbb{N}_{\text{expr}}}^n \times {\mathbb{N}_{\text{expr}}}^n$ *is the corresponding layout neutral form, where* G_n *is the set of* n*-dimensional array objects in P, and the two* ${\mathbb{N}_{\text{expr}}}^n$ *are the dimension vector and the index vector respectively.*

$\mathbb{N}_{\text{expr}}$ *is the union of natural numbers (*$\mathbb{N}$*) and expressions whose type is of* $\mathbb{N}$*.* ϵ *indicates the case where the triple is not defined for a given expression.*

In other words, LN maps an expression e to either a triple $(g \in G_n, \vec{D} \in {\mathbb{N}_{expr}}^n, \vec{I} \in {\mathbb{N}_{expr}}^n)$, where g, $\vec{D}$ and $\vec{I}$ are respectively an n-dimensional array, its dimension vector, and an index vector corresponding to the cell accessed by e, or ϵ when there is no such triple for e. In practice, the compiler must be able to map every array access expression of a particular array object to such a triple in order to insert the FF specific to that layout into each access expression. We loosely define a particular piece of code as "layout-neutral" if the compiler has all the information necessary to arbitrarily define the layout of the desired arrays.

D. Deriving Layout-Neutral Form from C Code

For C programs, we can derive layout neutral form by an informally specified LN from the type of operation of a given expression, as shown below:

- Fully-qualified array subscripting: we can derive $\vec{D}$ straightforwardly from the declaration of the array, and $\vec{I}$ from the expression. Consider the following C code snippet:

  ```
  float a[D][D];
  S1:   a[k+3j][i] = 1.0f;
  ```

  The layout neutral form of `a[k+3j][i]`:

  $$LN(\texttt{a[k+3j][i]}) = \begin{cases} (a, (D, D), (i, k+3j)) & \text{for } 0 \le i < D, \\ & 0 \le k+3j < D \\ \epsilon & \text{otherwise} \end{cases}$$

- Pointer dereferencing: we derive $\vec{I}$ by applying the inverse of RML on the offset to the base of the array. An example would be:

  ```
  float b[10][10][10]; int x;
  S1:   float *p = b;
  S2:   *(p + x) = 0; // e = *(p + x)
  ```

  The layout neutral form of expression e:

  $$LN(e) = \begin{cases} (b, (10, 10, 10), \vec{I}), & \vec{I} = RML_b^{-1}(p + x - b) \\ & 0 \le I_i < 10, 0 \le i < n \\ \epsilon & \text{otherwise} \end{cases}$$

- Other operations: ϵ

Conceptually the layout transformation process is to replace array accessing expression e with $FF(LN(e))$. For those expressions with pointer-derefencing e:`*(p+off)`, general we would substitute e with $FF(RML^{-1}(\texttt{off}))$. However, since the run-time cost of applying RML^{-1} to all array-accessing expressions might be prohibitive, it is generally preferred to derive layout-neutral form using the first method. That is, one implements all expressions accessing allocated multidimensional arrays using only fully-qualified subscripts in all dimensions to access elements.

E. Layout Transformations as Extended Types

Types in programming languages specify the information necessary for code to interpret and operate on the data 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. While this could be accomplished without compiler transformation by making the flattening function an indirect function associated with each array, these methods introduce undesirable overheads compared to static inlining of the flattening function.

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 which grid should the compiler perform automatic layout transformation, without specifying actual layout, and the compiler decides actual layout that works best on the given grid for the given architecture, and propagates this layout information through this analysis.

We shall describe this analysis as a monotonic dataflow analysis; in this framework a data-flow analysis is represented as a semilattice and a set of transfer functions. For this problem, the semilattice is (Ψ, $\wedge$), where each element in the semilattice is a function: $\Psi : P \rightarrow \mathbb{L} \cup \{UT, \perp\}$. $\mathbb{L}$ is a set of tuples representing the layout, where $(n \in \mathbb{N}, \mathbb{N}^n, \mathbb{N}^n \rightarrow \mathbb{N})$ denote dimension, dimension vector, and flattening function respectively. P is the set of pointer variables in the program, UT stands for *untransformed* and $\perp$ means *incompatible* respectively. An untransformed pointer indicates that the data structure pointed by this pointer 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 vector $\vec{D}$ and index vector $\vec{I}$, $FF_1(\vec{D}, \vec{I}) = FF_2(\vec{D}, \vec{I})$. 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 $f \in F; f : L \to L$ are created from the type of operations in the flow graph, as shown in Table I; the meet operation of two functions $m, n \in \Psi$ is defined in Table II. 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 $FF_1 == FF_2$.

TABLE I: Transfer Functions

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

TABLE II: Meet Operation

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

V. Directing Data Layout Transformation

This section describes how to derive the data layout to best match an application data structure to the MLP supported by the underlying memory hierarchy. Intuitively, the space of all possible layouts could be very large. In this section we demonstrate how an analytical model of the memory hierarchy and static analysis of the program can efficiently lead to an effective choice of layout. We derive the analytical model for an NVIDIA GeForce 280 GTX as an example, and use the computational model of that GPU to analyze the expected program execution and the concurrent requests likely to be generated.

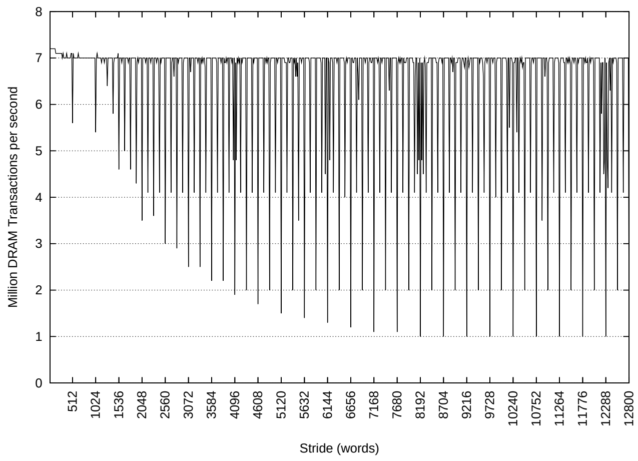

Fig. 3: Effective memory bandwidth v.s. strides in bytes between requests from from many single-threaded blocks on GTX280. Bandwidth is shown in millions of transactions per second, and strides are in increments of 64 bytes.

A. Benchmarking and Modeling Memory System Characteristics

A good layout should thus be able to exploit:

- Parallelism across different DRAM channels
- Parallelism across banks within a DRAM channel
- Pipeline parallelism within a DRAM burst

while preventing channel/bank imbalance by avoiding large strides among requests issued closely in time. In order to perform good data layout for structured grid applications, 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. Previous work [18] has benchmarked the GPU to obtain memory latency versus stride in a single-thread setting. However, since the class of applications we are targeting is mostly bandwidth-limited, we must determine how 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 [19] 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 SM.

Next, we must define our model on how bits in memory address steer interleaving among memory channels, DRAM banks, and any other parallel distribution structures that increases the number of requests that can be concurrently satisfied. We can determine these properties by generated

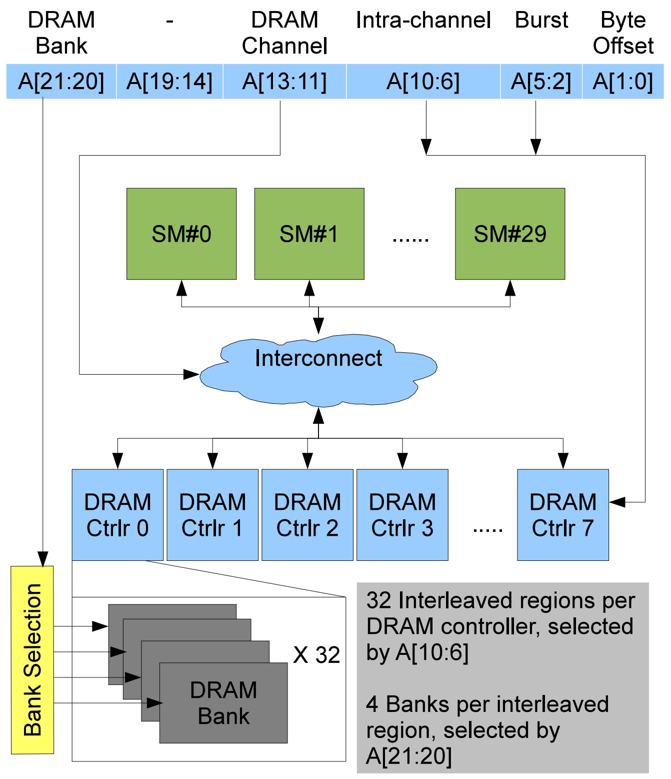

Fig. 4: DRAM Banking for GTX280 (Unit of Channel Interleaving = 2KB)

concurrent requests of a fixed stride pattern, and observing the resulting achieved bandwidth. Figure 3 shows how differences in the addresses of memory transactions affect memory bandwidth, in terms of million transaction per second. It is measured using a method similar to pointer-chasing in lmbench [20]: 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.

Figure 3 allows us to model how DRAM controllers are arranged on the GTX 280. Given that there are eight DDR3 DRAM controllers [21] we assume global memory is 8-channel interleaved. Also, we assume the DRAM burst size is equal to a coalesced 64-byte transaction. With these assumptions, we can approximate how memory accesses are interleaved across and within DRAM channels by comparing bandwidth for different power-of-two strides to obtain the degree of interleaving, and determining how memory banks are mapped within a channel by comparing bandwidth for strides with power-of-two ±1, ±2, and ±4 burst size (e.g. 16K+64 bytes stride). In general, larger offsets means a higher rate of hitting the same intra-channel region.

Since the access pattern for most 3D structured grid code will cause large strides, we are more interested in the interleaving scheme for large strides. For strides larger than 4KB, the interleaving unit for each DRAM channel is 2KB. For addresses falling in the same channel, it is interleaved internally with each of the 32 intra-channel regions capable of handling one distinct DRAM burst, as shown in Figure 4. Moreover, the DDR DRAM bank is decided by a different set of bits so even if two requests map to the same channel and same intra-channel port, they might still be able to proceed without interference by having distinct bank numbers. Thus, We define address bit $[13:6]$ as the *steering bits*, as those bits help decide the routing toward a specific bank.

B. Analysis of Application Memory Requests

Given a layout-neutral source code, the data layout tool must analyze the locality and distribution of index values that are likely to be requested concurrently or closely in time, to choose a layout that best distributes those indices among channels, ports and banks, while making as fully use as possibly of every burst. This is achieved by arranging bits in the index expressions known to generate unique, concurrent index values to those address bit fields that distribute accesses among parallel request handlers. In addition, the bits deciding the burst word offset must be generated from a SIMD instruction request, as explained in the coalescing rules [19].

The procedure of deriving a good layout for a given program is outlined as follows:

1) Classify index expressions based on their concurrent request distribution categories.
2) Compute the *span* of indices: determining the range of indices that will likely be concurrently requested.
3) Derive constraints from spans, coalescing rules, and DRAM banking revealed by micro-benchmarking
4) Solve the optimization problem for bits that should be tiled for all indices, subject to the constraints above
5) Derive the layout according to the optimal bit allocation obtained from the solution

The first step is to classify indices based on their run-time value ranges and distributions. Single-Program Multiple-Data parallel implementations will generally give rise three major kinds of index expressions.

- Category T: Indices that are computed with expressions varying among co-scheduled tasks, i.e. threads within a CUDA thread block These accesses are most important to the utilization of DRAM bursts, as these requests are typically generated very closely in time, or even simultaneously from parallel vector lanes. In CUDA, these are expressions dependent on `threadIdx`.
- Category B: Indices that are computed with expressions varying across collection of threads. Accesses of this category will be generated concurrently from different “cores” if collections of threads are issued to different execution resource. These are treated differently because the range of values of these kinds of indices tends to form a loose sliding window, as cores complete current collections of threads and retrieve others. Expressions dependent on `blockIdx.x` define this category in the CUDA execution model.
- Category I: Indices that do not belong to the above categories, and are either invariant among concurrent threads e.g. a fixed offset, or fully exercised by all threads

e.g. indices of different attributes of a grid cell. These two subtypes are separated later.

For structured-grid code implemented in LN-CUDA, the set of cells addressed by each thread is specified in terms of the thread index, block index, and/or thread-invariant expressions involving neither. For example, `A[blockIdx.x][threadIdx.x]`, is an index expression combining both block and thread indices to access a 2D grid `A`. Some applications, like LBM, may represent each grid cell with multiple scalar attributes addressed by indexing with another constant-sized grid dimension. If we take a snapshot during the execution of such programs, the distribution of index values of those grid dimensions issued by all active threads within the system would not directly depend on these thread's indices or the blocks they are grouped in. An index expression generally belongs to one or more categories above; e.g. an index like `threadIdx.x+blockIdx.x * blockDim.x` belongs to both Category T and Category B.

C. Span of Indices

The second step is to estimate the run-time value ranges, or *span*, for each category of index. For instance, Category I indices are separated into invariant and variant by analyzing their span. In CUDA, the span of Category T indices are simply determined by the kernel configuration parameters: all thread indices within a block are represented and scheduled concurrently to an SM. The span of concurrent block indices we can approximate from the block scheduling policy and number of concurrent blocks, described in more detail later. We do not expect strong correlation in the program counter values of threads in different thread blocks, so instances of Category I indices issued across CUDA warps usually don't have statically predictable locality. Hence, for Category I indices we assume a uniform distribution of possible index values.

For CUDA, spans for thread (Category T) and block (Category B) indices are defined as:

$$\begin{aligned}
span(\texttt{threadIdx.d}) &= \texttt{blockDim.d} : \texttt{d} \in \{\texttt{x}, \texttt{y}, \texttt{z}\} \\
span(\texttt{blockIdx.x}) &= \min(\#ActiveBlock, \texttt{gridDim.x}) \\
span(\texttt{blockIdx.y}) &= \frac{\#ActiveBlock}{gridDim.x}
\end{aligned}$$

Where $\#ActiveBlock$ is the number of blocks will be actively executing in the entire system, which can be determined statically from the compiled code's resource usage and the device parameters [19]. For example, when a kernel with $32 \times 4 \times 1$ thread block dimensions, 100×100 thread grid, executes on a device that can support 90 concurrent blocks, $span(\texttt{threadIdx.x}) = 32$, $span(\texttt{threadIdx.y}) = 4$ and $span(\texttt{threadIdx.z}) = 1$, $span(\texttt{blockIdx.x}) = 90$, and $span(\texttt{blockIdx.y}) = 1$.

As structured grid codes have a consistent set of classifications in indexing expressions in grid-accessing expressions, we can depict the layout-decision procedure as follows: given an index i extracted from grid-accessing expressions of a grid annotated for transformation, an ideal flattening would map as many lower $\log_2(span(i))$-bits as possible to the steering bits of a word address to reduce the chance of DRAM channel/bank conflict.

D. Data Layout Tiling as an Optimization Problem

In order to satisfy constraints imposed by memory hierarchy, we shall design a flattening function that effectively groups those parts from all dimensions that are likely to be concurrently indexed and form lower bits of flattened addresses from them. More precisely, we transform the flattening function RML by splitting indices with known "busy" parts: defining a bit of an index value as busy if it varies across concurrent dynamic instances that index expression, and shift those busy parts to the lower-order address bits by a series of index swaps, and the number of busy bits of an index $i \simeq \log_2(span(i))$. We can informally define this kind of *layout tiling* operation being applied on an index $I \in \vec{I}$ and a dimension $D \in \vec{D}$ as splitting I and D at bit position j and k where $0 \leq k < j$ and shift $I_{j:k}$ and $D_{j:k}$ to another position in the index and dimension vectors by a series of swaps with neighboring indices and dimensions.

We model the problem of deciding the number of index bits to be tiled for highly interleaved DRAM systems, such as the GPU global memory, as an optimization problem. The goal is to tile dimensions with spans larger than one, and make those tiled dimensions fit the DRAM channel and intra-channel interleaving characteristics as well as the DRAM burst size.

In following text, the variable representing the number of tiled bits of an index i is expressed as i_t:

maximize

$$\begin{aligned}
z = C_1\texttt{threadIdx.x}_t + C_2\texttt{threadIdx.y}_t &+ \\
C_3\texttt{threadIdx.z}_t + C_4\texttt{blockIdx.x}_t &+ \\
C_5\texttt{blockIdx.y}_t &
\end{aligned}$$

subject to

$$\begin{aligned}
p = \texttt{threadIdx.x}_t + \texttt{threadIdx.y}_t &+ \\
\texttt{threadIdx.z}_t + \texttt{blockIdx.x}_t &+ \\
\texttt{blockIdx.y}_t + span(I_t) &
\end{aligned}$$

and bounds of variables

$$\begin{aligned}
p &\leq \#SterringBits - \log_2\left(\frac{\texttt{sizeof}(T)}{\texttt{sizeof}(word)}\right) \\
\texttt{threadIdx.x}_t &\geq log_2(64/\texttt{sizeof}(T)) \\
\texttt{threadIdx.x}_t &\leq \lfloor log_2(span(\texttt{threadIdx.x}))\rfloor \\
0 \leq \texttt{threadIdx.y}_t &\leq \lfloor log_2(span(\texttt{threadIdx.y}))\rfloor \\
0 \leq \texttt{threadIdx.z}_t &\leq \lfloor log_2(span(\texttt{threadIdx.z}))\rfloor \\
0 \leq \texttt{blockIdx.x}_t &\leq \lfloor log_2(span(\texttt{blockIdx.x}))\rfloor \\
0 \leq \texttt{blockIdx.y}_t &\leq \lfloor log_2(span(\texttt{blockIdx.y}))\rfloor
\end{aligned}$$

where I_t represents the Cat. I index, if applicable.

The objective function means that we maximize the number of busy bits of indices tiled to the steering bits, according to weights representing the relative importance of Cat. T and Cat. B indices. The positive coefficients C_1 to C_3 depend on scheduling rules within an SM; C_4 and C_5 depend on how thread blocks are issued across SMs. Our heuristic sets $C_5, C_4 > C_1, C_2, C_3 > 0$, and they are fixed for a given architecture. This heuristic comes from an assumption that different thread blocks are more likely to be executed simultaneously than threads in the same block but in different warps, since thread blocks could be distributed to different SMs while threads in the same block but in different warps are executed in turns on the same scheduled SM. So we favor solutions that allocate more bits to block indices than to thread indices. This heuristic also assumes that block indices from thread block dimensions are enumerated and scheduled in row-major order.

The constraint models the interleaving constraint: since steering bits are bits 13-6 and bits 5-0 help form a DRAM burst, all tiled dimension should not go beyond that limit. We shall also tile entire $span(I_t)$ based on the assumption of uniform distribution of Category I indices, and expect those indices to be interleaved by the hardware.

The lower bound of *threadIdx.x* reflects how the global memory coalescing hardware works; we would tile lower-ordered bits of this index that would vary across threads in a CUDA warp to offsets within the 64-bytes segment.

The upper bounds of thread and block indices indicate the maximal degree of interleaving that each of those indices would need. Effectively, if at run time an index expression would have 2^b consecutive values, and that index is used to index a single grid dimension in all threads, then we should be able to interleave those requests with no more than 2^b different interleaved memory regions. For non-power-of-two spans, we take the floor in order to avoid under-utilization of some of those regions.

It is worth noting that given this particular GPU architecture and constraints obtained from modeling its memory interleaving, this optimization problem can be solved greedily as all the coefficients in objective function are positive. So starting from the index with largest coefficient in objective function, we tile all its bits until hitting the upper bound, then pick the second largest one, and so on.

E. Order of Tiled Indices

After the optimal bit allocation of each tiled index has been found, we arrange the dimensions in the transformed flattening function based on the following partial order:

$$\begin{aligned} &\top : \texttt{threadIdx.x}_L && (1)\\ Idx_H &\le Idx_L && (2)\\ \texttt{blockIdx.}d_p &\le \texttt{threadIdx.}d_p && (3)\\ CatI &\le Idx_L && (4)\\ \texttt{threadIdx.y}_p &\le \texttt{threadIdx.x}_p && (5)\\ \texttt{threadIdx.z}_p &\le \texttt{threadIdx.y}_p && (6)\\ \texttt{blockIdx.y}_p &\le \texttt{blockIdx.x}_p && (7) \end{aligned}$$

Where $Idx \in \{\texttt{blockIdx.}d, \texttt{threadIdx.}d\}, d \in \{x, y, z\}$, $p \in \{L, H\}$; Idx_H and Idx_L corresponds to $Idx_{[:Idx_t]}$ and $Idx_{[Idx_t-1:0]}$, respectively if $Idx_t > 0$. $CatI$ refers to Category I indices.

Order 1 reflects the hardware limitation that in order to get full coalescing within a half-warp, *threadIdx.x* must be tiled as the lowest bits of the flattened offset. Order 2 is obvious: "busy" bits of an index should be mapped to lower bits in flattened offset than their counterparts. Order 3 to 7 are the heuristics we employ:

- Arrange thread indices first, then block indices. Requests from different SMs are thus likely to be issued to different DRAM channels, rather than to same channel but different ports.

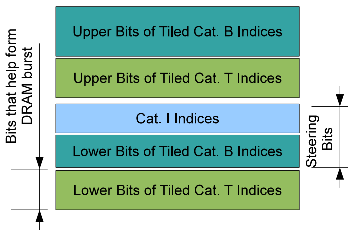
Bits that help form DRAM burst
Upper Bits of Tiled Cat. B Indices
Upper Bits of Tiled Cat. T Indices
Cat. I Indices
Lower Bits of Tiled Cat. B Indices
Lower Bits of Tiled Cat. T Indices
Steering Bits

Fig. 5: Typical Data Layout in terms of Index Classes

- Tile Category I indices after tiling all tiled thread/block indices, because Category I indices are not functions of thread/block indices and their value distributions are more likely to depend on program counter. It is thus considered less likely to be contiguous across requests and should not be tiled to the bit range that is being used by intra-channel interleaving.
- For thread indices, tile in the warp forming order.
- Assuming row-majored thread block issuing order, tile the X dimension first.

A tiling scheme that follows those layout tiling rules of the index classes may look like Figure 5. This figure represents how individual bits in the flattened offset come from different classes of indices.

F. An Example of Data Layout Transformation

The following example is an LN-CUDA implementation of a 7-point stencil code kernel that solves a 3D heat equation [22];

Listing 1: 7-pt 3D heat solver in LN-CUDA

```
// Declare A0 and Anext as 3D
// variable-length arrays
__global__ void heat_kernel(float fac,
    int nx, int ny, int nz,
    float A0[nz][ny][nx],
    float Anext[nz][ny][nx])
{
  int i = threadIdx.x+1, j = blockIdx.x+1;
  int k = blockIdx.y+1;

  // Access in FORTRAN-like form which
  // is then converted to LNF
  Anext[k][j][i] = 0.8f/6.0f * (
    A0[k + 1][j][i] + A0[k - 1][j][i] +
    A0[k][j + 1][i] + A0[k][j - 1][i]] +
    A0[k][j][i + 1] + A0[k][j][i - 1] )
    + 0.2f * A0[k][j][i];
}
```

And given the following thread grid information:

$$\texttt{blockDim.x} = 254;\ \texttt{gridDim.x} = 254$$
$$\texttt{gridDim.y} = 126;\ Occupancy = 1.0$$

The spans of thread grid indices are then computed:

$$\begin{aligned} span(\texttt{threadIdx.x}) &= 254\\ span(\texttt{blockIdx.x}) &= (\#\textit{thread blocks/SM} \times \#SM)\\ &\quad \%\texttt{gridDim.x} = 120\\ span(\texttt{blockIdx.y}) &= 1 \end{aligned}$$

So the three subscript expressions used to index `A0` and `Anext` are of Category-B, Cat. B, and Cat. T respectively, with their spans being 1, 120, and 254. The solution of the optimization problem stated in section V-D suggests tiling

7 bits of the x dimension, and 5 bits of the y dimension. After applying layout tiling according to the tiling above, the resulting dimension vector $\vec{D}$ and flattening function FF for `Anext` and `A0` will be:

$$\vec{D} : (\mathtt{nz}, \lceil \mathtt{ny}/2^5 \rceil, \lceil \mathtt{nx}/2^7 \rceil, 2^5, 2^7)$$

$$\begin{aligned} FF(\vec{I}, \vec{D}) : \quad & I_2 \times \lceil \mathtt{ny}/2^5 \rceil \times \lceil \mathtt{nx}/2^7 \rceil \times 2^5 \times 2^7 \\ & + \; I_{1[:5]} \times \lceil \mathtt{nx}/2^7 \rceil \times 2^5 \times 2^7 \\ & + \; I_{0[:7]} \times 2^5 \times 2^7 + I_{1[4:0]} \times 2^7 \\ & + I_{0[6:0]} \end{aligned}$$

VI. Experimental Results

Table III shows relative speedups of different memory layouts. LBM is a CUDA implementation of SPEC CPU2006 [23] 470.LBM, which implements lattice-Boltzmann method [8]; CFD is a kernel that performs either a red or black sweep using Gauss-Seidel method; This kernel is from CU-FLOW, a 3D Navier-Stokes equation solver. Heat implements a 3D heat equation solver using the Jacobi scheme, as described in [12].

The last two benchmarks represent the two major point methods for solving PDEs using the finite difference method. LBM is an alternative CFD approach that uses particle-based method instead of discretizing the PDE.

For each of the benchmarks, we first manually convert them into layout-neutral form and apply our layout transformation methodology on the main grids on which each benchmark operates. Because our compiler infrastructure does not yet support variable-length array syntax, we use annotations to communicate that information to the compiler. Then in the potential layout transformation space, manual search is applied to nearby regions on the solution found by our methodology; we then compare the baseline layout of each benchmark with two layouts.

TABLE III: Benchmarks and Speedup

Benchmark	Layout	Speedup	Bandwidth (GB/s)
LBM	Array of Structures	1.0	18.10
	Structures of Array	5.11	92.60
	Transformed (auto)	6.60	119.50
	Transformed (manual)	6.60	119.50
CFD	Row Major Layout	1.0	56.63
	Transformed (auto)	1.25	70.93
	Transformed (manual)	1.30	73.75
Heat	Row Major Layout	1.0	74.31
	Transformed (auto)	1.07	79.37
	Transformed (manual)	1.08	80.17

Significant speedups are observed from all benchmarks, ranging from 6.6X (LBM) to 1.07X (Heat) with automatically derived layout. The performance different between layout-optimized and baseline layout is tied to how far the transformed layout diverges from the baseline. For instance, the transformed LBM layouts more closely resemble a (tiled) structure of array form than array of structure form: much of the performance is gained from improved burst-level parallelism due to improved memory coalescing, and swapping the structure field index into a higher bit position. We earn 30% additional performance gained from tiling by making busy bits stay in steering bit position. On the other extreme, the Heat benchmark's transformed layout is actually very close to RML, with only a very small degree of tiling in x and y introduced. The performance therefore only increases slightly, as the original layout was quite good for that particular memory system.

The achieved bandwidth of the transformed code is primarily dependent on the proportion of accesses that are aligned in the transformed layout. It is impossible to force all accesses to be aligned, as many stencil codes will generate indices of i, i+1, and i-1, which cannot all be aligned in any layout. For our system, unaligned accesses essentially use twice the bandwidth necessary, as two bursts are triggered for only one burst-size of unaligned data. Of our benchmarks, LBM has the lowest percentage of unaligned accesses, with 29 aligned and only 10 unaligned accesses per thread. With the inherent waste of partially used bursts, the transformed LBM has a hard theoretical bandwidth limit of 87% of peak. The achieved bandwidth of the transformed LBM represents 85% of peak bandwidth, or 98% of the theoretical bandwidth limit given partial-burst constraints. The other two applications have a much higher proportion of unaligned accesses, resulting in lower achieved bandwidths.

Also, our experiment shows that even with extra overhead computing memory addresses, the transformed applications still gained performance by improving the efficiency of the memory hierarchy. This highlights both the bandwidth-boundedness of the applications themselves, and the validity of trading extra address calculation instructions for better achievable bandwidth in such bandwidth-bound situations.

VII. Conclusion and Future Work

We have presented a formulation and language extension that enables data layout transformation for structured grid codes in CUDA. We also benchmarked the GTX280 GPU to reveal its DRAM banking and interleaving scheme. Based on the micro-benchmark 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 aims at improving reuse. Future work investigating holistic data layout transformations addressing temporal locality, spatial locality, and MLP will be 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. We would like to thank Chris Rodrigues and Nady Obeid for their comments.

REFERENCES

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, “The landscape of parallel computing research: A view from berkeley,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[2] K. W. Morton and D. F. Mayers, *Numerical Solution of Partial Differential Equations: An Introduction*. New York, NY, USA: Cambridge University Press, 2005.

[3] J. H. Ferziger and M. Peric, *Computational Methods for Fluid Dynamics*. Berlin: Springer, 1999.

[4] C. D. Gundolf, C. C. Douglas, G. Haase, J. Hu, M. Kowarschik, and C. Weiss, “Portable memory hierarchy techniques for PDE solvers, part II,” *SIAM News*, vol. 33, pp. 8–9, 2000.

[5] J. W. Demmel, *Applied numerical linear algebra*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1997.

[6] Y. H. Qian, D. D’Humieres, and P. Lallemand, “Lattice BGK models for Navier-Stokes equation,” *Europhysics Letters*, vol. 17, no. 6, pp. 479–484, 1992.

[7] Y. Zhao, “Lattice Boltzmann based PDE solver on the GPU,” *Visual Computing*, vol. 24, no. 5, pp. 323–333, 2008.

[8] T. Pohl, M. Kowarschik, J. Wilke, K. Iglberger, and U. Rüde, “Optimization and profiling of the cache performance of parallel lattice boltzmann codes,” *Parallel Processing Letter*, vol. 13, no. 4, pp. 549–560, 2003.

[9] G. Rivera and C.-W. Tseng, “Tiling optimizations for 3D scientific computations,” in *SC00: Proceedings of the 2000 conference on Supercomputing*, 2000, p. 32.

[10] S. Sellappa and S. Chatterjee, “Cache-Efficient Multigrid Algorithms,” *International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 115–133, 2004.

[11] Q. Lu, C. Alias, U. Bondhugula, T. Henretty, S. Krishnamoorthy, J. Ramanujam, A. Rountev, P. Sadayappan, Y. Chen, H. Lin, and T.-f. Ngai, “Data layout transformation for enhancing data locality on nuca chip multiprocessors,” in *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, 2009, pp. 348–357.

[12] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, “Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures,” in *SC08: Proceedings of the 2008 conference on Supercomputing*, Piscataway, NJ, USA, 2008, pp. 1–12.

[13] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, “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*, 2008, pp. 73–82.

[14] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, “Self-optimizing memory controllers: A reinforcement learning approach,” *Computer Architecture News*, vol. 36, no. 3, pp. 39–50, 2008.

[15] T. Moscibroda and O. Mutlu, “Distributed order scheduling and its application to multi-core DRAM controllers,” in *Proceedings of the 27th Symposium on Principles of Distributed Computing*, 2008, pp. 365–374.

[16] O. Mutlu and T. Moscibroda, “Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems,” *Computer Architecture News*, vol. 36, no. 3, pp. 63–74, 2008.

[17] J. Shao and B. T. Davis, “A burst scheduling access reordering mechanism,” in *Proceedings of the 13th International Symposium on High Performance Computer Architecture*, 2007, pp. 285–294.

[18] V. Volkov and J. W. Demmel, “Benchmarking gpus to tune dense linear algebra,” in *SC08: Proceedings of the 2008 conference on Supercomputing*, 2008, pp. 1–11.

[19] nVIDIA, “nvidia cuda programming guide 2.0,” 2008.

[20] L. McVoy and C. Staelin, “lmbench: portable tools for performance analysis,” in *Proceedings of the 1996 USENIX Annual Technical Conference*, 1996, pp. 23–23.

[21] nVIDIA, “nvidia geforce gtx 200 gpu architectural overview,” 2008.

[22] S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. Yelick, “Impact of modern memory subsystems on cache optimizations for stencil computations,” in *Proceedings of the 2005 workshop on Memory system performance*, 2005, pp. 36–43.

[23] C. D. Spradling, “Spec cpu2006 benchmark tools,” *Computer Architecture News*, vol. 35, no. 1, pp. 130–134, 2007.