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

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

Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign
{sung10, stratton, w-hwu}@illinois.edu

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 have enabled 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 560% performance increases over the language-defined layout, and a 7% performance gain in the worst case, in which the language-defined layout and access pattern is already well-vectorizable by the underlying hardware.

Categories and Subject Descriptors

E.1 [**Data Structures**]: Arrays; C.1.2 [**Processor Architecture**]: Multiple Data Stream Architectures; E.2 [**Data Storage Representations**]: Contiguous representations; D.3.4 [**Processors**]: Compilers

General Terms

Performance, Measure, Design

Keywords

GPU, Parallel programming, Data layout transformation

PACT'10, September 11–15, 2010, Vienna, Austria.

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 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. When parallelizing such applications, the most common approach is spatial partitioning of the grid cell computations into fixed-size portions, usually in the shape of planes or cuboids, assigning portions 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; 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 developed by previous efforts to achieve vectorization [11] or for reducing coherence overhead [12].

A less explored direction is the parallelism among memory controllers, and interleaved DRAM banks, which is playing an increasingly important role in system performance. In massively parallel systems, the interconnect which connects DRAM channels and processors decodes address bit fields to decide corresponding channel and memory bank numbers from a memory request [3]. Given that a fixed subset of the address bits are used to spread accesses across parallel memory channels and banks, achieving high bandwidth requires that concurrently serviced accesses have varying values in those address bit fields. To exploit this level of memory-

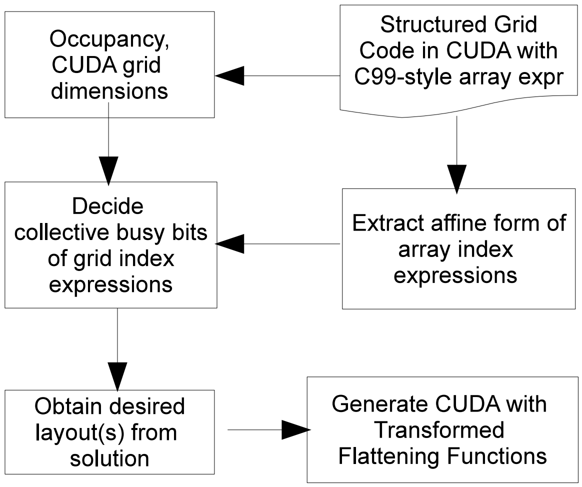

Figure 1: Data Layout Transforms for Structured Grid Codes

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 achieve a shuffling of the address bit fields such that concurrent memory requests can be both routed to different memory channels and banks are well-vectorized.

Unfortunately, the full details of the memory hierarchy are often too obscure or complex for a typical application programmer to adapt their programs to use them. Even for 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.

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 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. To make data layout transformation feasible in the context of a language derived from C++, we rely on the two assumptions. First the declaration, allocation, and access of multidimensional arrays should follow C99-style variable length array (VLA) syntax, and second the programmer must adhere to the FORTRAN-style subscripted array accesss, as any assumptions on the relation between addresses across multiple array elements may not hold after layout transformation.

Figure 1 depicts our procedure of data layout transformation, using a modern GPU as an example memory system. 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 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 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 4.1 discusses how we obtained a memory address interleaving scheme of the DRAM controller through microbenchmarking, and derive an optimized layout from the program and execution model. Section 6 presents our experiment results, followed by some concluding remarks in Section 7.

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 Jacobi method and Gauss-Seidel (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 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) [25], a particle-based method that was mainly used in computational fluid dynamics problems, was recently extended as a general PDE solver [32]. 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 [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 within an update iteration across cells there is no data reuse, techniques that aim at reducing memory accesses such as shared memory tiling for the GPU is 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 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 [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 use 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].

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 [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 primarily been used for improving cache locality and localizing memory accesses in nonuniform memory architectures and clusters. Anderson et al. [1] employed similar data layout transformations for shared-memory multiprocessor system to make the data accessed by the same processor contiguous in the address space. Lu et al.'s recent work [16] applies data layout transformation for cache locality in NUCA (non-uniform cache architecture) chip multiprocessors to keep memory access 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 to reduce DRAM bank conflicts, with the target architecture being many-cores connecting to on-chip multi-channel memory controllers through on-chip interconnect 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 access for clusters and NUCA. However, for a massively parallel system, balancing DRAM traffic across controllers can be important. Datta et al. [5] take into consideration the affinity of DRAM controller 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, 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 gaining unit-strided accesses [27, 11], 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 organization. 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 constraints and hence not able to further increase memory access efficiency on strided accesses commonly seen in LBM and red-black Gauss-Seidel methods codes.

Various works[10, 21, 22, 29] have proposed hardware optimizations for DRAM controllers working toward uniform access latency and fairness, some parallelism aware [21, 22]. All of them focus on scheduling DRAM requests under fixed workloads, not 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 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* $\mathbb{N}^n$ *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:* $\mathbb{N}^n \rightarrow \mathbb{N}$ *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 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

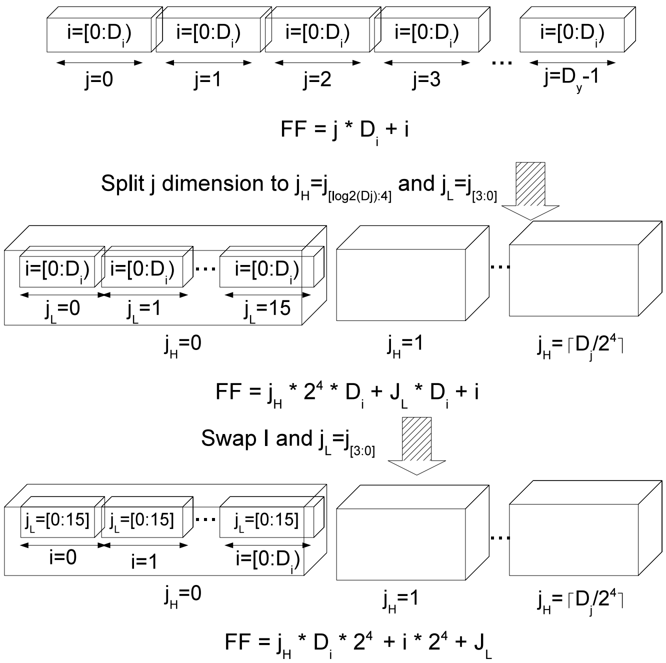

Figure 2: An Example of Layout Transformation

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.

To permute the address bit pattern derived by an FF, we can transform the 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 $\vec{I}'$ 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 $I_h = \lfloor I_i/T \rfloor$, $I_l = I_i \bmod T$ and $D_h = \lceil D_i/T \rceil$, $D_l = T$. Intuitively the strip-mining splits the dimension into two adjacent dimensions. When the original dimension size is not a multiple of block size, padding is introduced at the last block.

Permutation: Permute the index vector and corresponding dimension vector.

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 $\texttt{A}[\texttt{j}_{\log 2(\texttt{D}_i):4}][\texttt{i}][\texttt{j}_{3:0}]$. 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.

5. DIRECTING DATA LAYOUT TRANSFORMATION

Intuitively, the space of all possible layouts that can be derived by arbitrary application of the data layout transformation primitives on a multi-dimensional data structure could be very large; however by leveraging properties from both the SPMD programming model which is commonly seen on massively parallel systems and the class of application 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 code with large datasets.

In such systems, the interconnect and DRAM controllers spread these concurrent requests into different memory channels and banks mostly by hashing address bits; moreover, on some systems such as the NVIDIA G80 and GT200 series 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 the 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 the analytical model for an NVIDIA GeForce 280 GTX, and use the computational model of that GPU to analyze the expected program execution and the concurrent requests likely to be generated. Other devices and programming models could be evaluated independently with a similar approach. Previous work [31] 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 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 SM.

Next, we must define our model on which bits in memory address steer interleaving among memory channels, DRAM banks, or other parallel distribution structures built into the architecture to increases the number of concurrently satisfiable requests. We can determine these *steering bits* by observing the behavior of a microbenchmark generating concurrent requests of a fixed stride pattern and observing the resulting achieved bandwidth. 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-performing bandwidth in Figure 3, we can see a couple of principles 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 the lower address bits. Continued performance degradation as the stride doubles indicates that the bit that was variant in the previous power of two but static in the current one was relevant to the parallel distribution of requests. Figure 3 shows strides of 512, 1024, 2048, 4096 and 8192 words achieve successively lower effective bandwidth. 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. Second: 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 achieved MLP. For instance, note that strides of 8192+x*512 words are equally poor in performance as 8192 strides. Through further detailed micro-benchmarking, 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 Two-level SPMD Programming Model

In GPU architectures, each thread can only execute one memory operation at 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 variation in the values of those index expressions, and therefore variation in the bits representing the resulting address. To achieve the highest bandwidth, the intuitive goal of data layout transformation is to ensure that the address bits highly 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 following CUDA-like pseudo code that is a simplified version of 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 structure

__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-structure 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 expressions 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-structure layout is good for CPU or cache-based architecture 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 request a certain field of their own cell. The coalescing rules effectively state that the index of the lowest dimension must be dependent on `threadIdx` for good coalescing. This issue is easily relieved by permuting the data layout, perhaps by exchanging the second and last dimensions, as the higher dimensional indices are all functions of block index or values independent of block or thread indices, all higher-order bits of addresses to be produced by adjacent threads in a warp will lead 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 having distinct bits at those steering bits. Intuitively we should not only make 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. Furthermore, since program counters are not strongly correlated across blocks, the index of the lowest dimension will also likely vary across concurrent requests, as that index is tied to the program counter. A good data layout would take these *busy bits* from the index of each dimension and map those bits into the steering bits of the memory system. A more formal definition and automated solution is presented in the remained of this section.

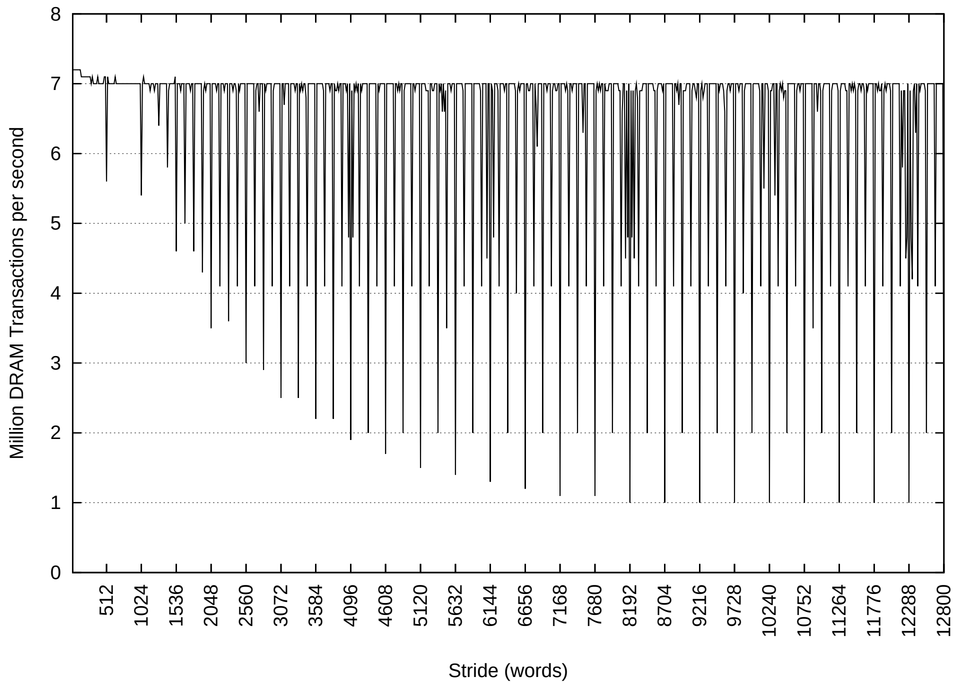

Figure 3: Effective memory bandwidth v.s. strides in bytes between requests from from many single-threaded blocks on GTX280. Bandwidth is shown in millions of transactions per second, and strides are in increments of 64 bytes.

5.2.1 Characterizing Thread Indices in Two-level SPMD Programming Models

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

- The computational grids consist of fix-sized blocks of threads issued as a whole to the processors (i.e. SM in CUDA terms). Distinct blocks are executed asynchronously across processors. For thread IDs, because of the asynchronism any thread with 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 ID usually is only a fraction of total number of blocks in a computational grid. With some simplifying assumptions about the regularity of block execution time, the index range of currently executing blocks 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 concurrent instances of them:

- The number of distinct least significant bits across concurrent block IDs is about $\log_2$ (maximum capacity for 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 thread block index will be busy and 7 LSBs of thread index will be busy. In this case one good layout for array `A0` could be created by strip-mine y and x dimension by 32 and 128 respectively and shift subdimensions into steering bit position and the bit fields that is used for memory coalescing. In terms of dimension vector and flattening function, the dimension vector of `A0` is $\vec{D} : (\lceil \texttt{ny}/2^5 \rceil, \lceil \texttt{nx}/2^7 \rceil, 4, 2^5, 2^7)$, where `nx` and `ny` are from C99 VLA declaration of `A0`; the FF of `A0` is $FF(\vec{I}, \vec{D})$: $I_{2[:5]}D_3D_2D_1D_0+I_{1[:7]}D_2D_1D_0+I_0D_1D_0+I_{2[4:0]}D_0+I_{1[6:0]}$, where $\vec{I}$ is the index vector of the array subscripts, e.g. for `A0[j][i][0]`, $\vec{I} : (I_2 = j, I_1 = i, I_0 = 0)$.

5.2.2 Automated Discovery of Ideal Data Layout

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, usually the array-accessing expressions can be converted to this form. In principle, if there are non-affine terms in the ex-

pression, we could still approximate it by introducing auxiliary affine terms, as suggested by Girbal et al [8].

2. For a given grid, if all the expressions accessing the grid share the same coefficient 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 coefficients of affine form of accessing expressions except the constant column as the grid's *common access pattern*. For structured grid codes which accesses nearest neighbors, the access pattern to the same grid usually have the same coefficient except for the last column. E.g. `[x+1][y]` and `[x-1][y-1]` is considered of the same common access pattern $\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}\begin{pmatrix}\mathtt{x}\\\mathtt{y}\end{pmatrix}$.

3. For each eligible grids, derive the desired data layout from its common access pattern.
   (a) Decide number of busy bits of each referred thread and block index from occupancy and thread block configuration.
   (b) For each dimension compute the collective busy bits represented by the corresponding row in common access pattern. Since a row in 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 of all dimensions to the steering bits by using strip-mining of power-of-two-sized tiles and permuting these tiles to the desired bit position until steering bits are all occupied or there are no busy bits left for any dimension.
   (e) Assign all unassigned dimensions to higher dimensions.
   (f) Generate flattening functions and dimension vector according to above assignment and the C99 VLA declaration for the grid.

4. Perform the data-flow analysis to derive the flattening function associated with each array accessing expressions.

5. Output transformed the code with inline-expanded flattening function at grid accessing expressions.

For our example, we can list some of the access functions of `A0` and `Anext`: $(blockIdx.x+1, threadIdx.x+1, E)$, $(blockIdx.x+1, threadIdx.x+1, W)$, $(blockIdx.x, threadIdx.x+1, N)$, and $(blockIdx.x+2, threadIdx.x+1, S)$. In affine form of access function similar to the notation used by Girbal et al, they would look like: $\begin{bmatrix}1 & 0 & 1\\0 & 1 & 1\\0 & 0 & \mathtt{E}\end{bmatrix}\begin{pmatrix}\mathtt{blockIdx.x}\\\mathtt{threadIdx.x}\\1\end{pmatrix}$, $\begin{bmatrix}1 & 0 & 1\\0 & 1 & 1\\0 & 0 & \mathtt{W}\end{bmatrix}\begin{pmatrix}\mathtt{blockIdx.x}\\\mathtt{threadIdx.x}\\1\end{pmatrix}$, $\begin{bmatrix}1 & 0 & 0\\0 & 1 & 1\\0 & 0 & \mathtt{N}\end{bmatrix}\begin{pmatrix}\mathtt{blockIdx.x}\\\mathtt{threadIdx.x}\\1\end{pmatrix}$, and $\begin{bmatrix}1 & 0 & 2\\0 & 1 & 1\\0 & 0 & \mathtt{S}\end{bmatrix}\begin{pmatrix}\mathtt{blockIdx.x}\\\mathtt{threadIdx.x}\\1\end{pmatrix}$, respectively. So accesses to both `Anext` and `A0` are eligible for transformation as the affine form of their access functions only differ in the last column, and these arrays are all 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 then 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 examples, we will assume the number of active thread block 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 threadIdx.x, takes the lowest dimension place in the transformed layout for coalescing (4 least significant bits) and the first three steering bits. The highest dimension is tiled split into two dimensions, with the 5 low bits accesses 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 Propagate 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 next iteration, i.e. the output of current iteration become 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 layout as extended types and solve the data-flow equation to analyze the layout for array accessing expressions.

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. This propagation could be performed at runtime by extending the array type in the compiler to augment the grid with an function pointer to the flattening function, set when the array is allocated. However, current GPU programming models do not allow indirect calls; 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,

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 $\nu(\texttt{p1}) = ...$, 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	$\nu(\texttt{p1}) = \bot$ 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	$\bot$
`l2`	if `l1` == `l2` then `l1` else $\bot$	$\bot$	$\bot$
UT	$\bot$	UT	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$

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.

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 data-flow 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: $\Psi : P \to \mathbb{L} \cup \{UT, \bot\}$. P is the set of pointer variables in the program, UT stands for *untransformed* and $\bot$ 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 $\mathbb{N}^n \to \mathbb{N}$). 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 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 $f : \Psi \to \Psi$ are created from the type of operations 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. Transformed and untransformed layouts, as well as dimension vectors of grids, are thus propagated.

The meet function $\wedge$ deals with the join of control flow. 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 more than one flatten functions that need to bind with any expression that accesses the grid and these functions are incompatible (i.e. the binary relation == does not hold for these functions). This restriction can surely be slightly relaxed by using versioning, but this is left for future work.

6. EXPERIMENTAL RESULTS

Table 3 shows relative speedups of different memory layouts. LBM is a CUDA implementation of SPEC CPU2006[30] 470.LBM, which implements lattice-Boltzmann method[24]; 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 [5] implements a 3D heat equation solver using the Jacobi scheme.

The last 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 of the benchmarks, we first manually convert them into layout-neutral form and apply our automated layout transformation methodology on the main grids on which

Table 3: **Benchmarks and Speedup**

Benchmark	Layout	Speedup	Consumed Bandwidth	Global Memory Reads per DRAM Transaction
LBM	Array of Structures	1.0	17.56 GB/s	1.01
	Structures of Array	5.11	89.87 GB/s	6.60
	Transformed (auto)	6.60	115.99 GB/s	7.89
	Transformed (manual)	6.60	115.99 GB/s	7.89
CFD	Row Major Layout	1.0	56.63 GB/s	2.61
	Transformed (auto)	1.25	70.93 GB/s	3.14
	Transformed (manual)	1.30	73.75 GB/s	3.14
Heat	Row Major Layout	1.0	74.31GB/s	3.79
	Transformed (auto)	1.07	79.37 GB/s	3.79
	Transformed (manual)	1.08	80.17 GB/s	3.79

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. After automatic transformation, manual search is applied to nearby regions of the space of potential layout transformations near the solution found by our automated methodology, choosing the best. Table 3 shows the comparison of the automated and manual layouts against the baseline layout of each benchmark.

The effectiveness of data layout transformation on improving memory coalescing is measured in the column "Global Memory Reads per DRAM Transaction" [1] in Table 3. Larger number means that the memory coalescing hardware were able to vectorize more accesses from the same CUDA warp into a single DRAM transaction. Poor coalescing leads to lower number of effective global memory loads per measured coalesced DRAM transaction. The baseline LBM application shows the lowest global memory load coalesced per DRAM transaction, whereas the optimized LBM implementations ultimately show the best out of the benchmarks. 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, the best transformed LBM code has the lowest percentage of unaligned accesses, with 29 aligned and only 10 unaligned accesses per thread. From CFD, transformed layouts lead to higher global memory read per DRAM transaction.

In Heat and CFD, we can see that even with the memory coalescing level staying constant, bandwidth usage is increasing with effective tiling for memory interleaving hardware. For instance, the performance of Heat is improved by 7% with layout transformation, even though coalescing was not improved.

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 best 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 positions as well as improved memory coalescing. On the other extreme, the Heat benchmark's best transformed layout is 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.

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.

[1] This is the number of memory accesses per grid cell in the source code, times the number of grid elements divided by number of SMs in the system (30 for GTX280), and then divided by `gld_coherent` reported by the CUDA Profiler

7. 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.

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.

8. REFERENCES

[1] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam. Data and computation transformations for multiprocessors. *SIGPLAN Not.*, 30(8):166–178, 1995.

[2] 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. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[3] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *ISPASS*, pages 163–174. IEEE, 2009.

[4] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. A compiler framework for optimization of affine loop nests for gpgpus. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 225–234, New York, NY, USA, 2008. ACM.

[5] 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*, pages 1–12, Piscataway, NJ, USA, 2008.

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

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

[8] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Int. J. Parallel Program.*, 34(3):261–317, 2006.

[9] 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*, 33:8–9, 2000.

[10] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. *Computer Architecture News*, 36(3):39–50, 2008.

[11] B. Jang, P. Mistry, D. Schaa, R. Dominguez, and D. Kaeli. 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*, pages 353–354, New York, NY, USA, 2010. ACM.

[12] Y.-L. Ju and H. G. Dietz. 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*, pages 344–358, London, UK, 1992. Springer-Verlag.

[13] K. Kennedy and U. Kremer. Automatic data layout for high performance fortran. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 76, New York, NY, USA, 1995. ACM.

[14] V. Kindratenko, J. Enos, and G. Shi. Gpu clusters for high-performance computing. In *Proceedings of the Workshop on Parallel Programming on Accelerator Clusters*, Jan 2009.

[15] Y.-S. Kwon, B.-T. Koo, and N.-W. Eum. 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*, pages 329–334, Piscataway, NJ, USA, 2009. IEEE Press.

[16] 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*, pages 348–357, 2009.

[17] M. E. Mace. *Memory Storage Patterns in Parallel Processing*. Kluwer Academic Publishers, 1987.

[18] N. R. Mahapatra and B. Venkatrao. The processor-memory bottleneck: problems and solutions. *Crossroads*, 5(3es):2, Apr 1999.

[19] L. McVoy and C. Staelin. lmbench: portable tools for performance analysis. In *Proceedings of the 1996 USENIX Annual Technical Conference*, pages 23–23, 1996.

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

[21] 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*, pages 365–374, 2008.

[22] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. *Computer Architecture News*, 36(3):63–74, 2008.

[23] nVIDIA. nvidia cuda programming guide 2.0, 2008.

[24] 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*, 13(4):549–560, 2003.

[25] Y. H. Qian, D. D'Humieres, and P. Lallemand. Lattice BGK models for Navier-Stokes equation. *Europhysics Letters*, 17(6):479–484, 1992.

[26] G. Rivera and C.-W. Tseng. Tiling optimizations for 3D scientific computations. In *SC00: Proceedings of the 2000 conference on Supercomputing*, page 32, 2000.

[27] 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*, pages 73–82, 2008.

[28] S. Sellappa and S. Chatterjee. Cache-Efficient Multigrid Algorithms. *International Journal of High Performance Computing Applications*, 18(1):115–133, 2004.

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

[30] C. D. Spradling. Spec cpu2006 benchmark tools. *Computer Architecture News*, 35(1):130–134, 2007.

[31] V. Volkov and J. W. Demmel. Benchmarking gpus to tune dense linear algebra. In *SC08: Proceedings of the 2008 conference on Supercomputing*, pages 1–11, 2008.

[32] Y. Zhao. Lattice Boltzmann based PDE solver on the GPU. *Visual Computing*, 24(5):323–333, 2008.