

GPUDrano: Detecting Uncoalesced Accesses in GPU Programs

Rajeev Alur, Joseph Devietti, Omar S. Navarro Leija and Nimit Singhania

University of Pennsylvania



Abstract. Graphics Processing Units (GPUs) have become widespread and popular over the past decade. Fully utilizing the parallel compute and memory resources that GPUs present remains a significant challenge, however. In this paper, we describe GPUDrano: a scalable static analysis that detects uncoalesced global memory accesses in CUDA programs. Uncoalesced global memory accesses arise when a GPU program accesses DRAM in an ill-structured way, increasing latency and energy consumption. We formalize the GPUDrano static analysis and compare it empirically against a dynamic analysis to demonstrate that false positives are rare for most programs. We implement GPUDrano in LLVM and show that it can run on GPU programs of over a thousand lines of code. GPUDrano finds 133 of the 143 uncoalesced static memory accesses in the popular Rodinia GPU benchmark suite, demonstrating the precision of our implementation. Fixing these bugs leads to real performance improvements of up to 25%.

1 Introduction

Graphics Processing Units (GPUs) are well-established as an energy-efficient, data parallel accelerator for an increasingly important set of workloads including image processing, machine learning, and scientific simulations. However, extracting optimal performance and energy efficiency from a GPU is a painstaking process due to the many sharp corners of current GPU programming models. One particularly sharp corner arises when interacting with the memory hierarchy. We propose the GPUDrano system, the first scalable static analysis to identify an important class of memory hierarchy performance bugs for GPU programs. To show what GPUDrano does, we first explain a bit about the GPU memory hierarchy and the specific class of performance bugs, known as *global memory coalescing bugs*, that GPUDrano targets.

Load and store instructions that reference the GPU’s DRAM (known as *global memory*) must obey a certain structure to ensure that memory bandwidth is fully utilized. Accesses that do not exhibit this structure result in underutilization and can lead to significant performance problems. When a GPU program executes a load or store instruction, the memory address(es) referenced are mapped to aligned 128-byte cache blocks [18, §5.3.2], which is the physical granularity at which DRAM is accessed. GPUs bundle multiple threads together for single-instruction multiple-data (SIMD) execution, and we say that a SIMD load/store

is *coalesced* if its accesses are contained within a single cache block, otherwise the access is *uncoalesced*. Uncoalesced accesses are difficult to spot, even for seasoned GPU developers, and in some cases rewriting can avoid such uncoalescing, as is the case for many of our benchmarks from an established benchmark suite.

```
int array[N]; // in global memory

int a = array[0]; // Example 1: coalesced
int b = array[tid]; // Example 2: coalesced
int c = array[8*tid]; // Example 3: uncoalesced
```

Fig. 1: Examples of coalesced and uncoalesced memory accesses.

Figure 1 shows simple examples of coalesced and uncoalesced memory accesses. Each thread in the program executes the code shown (we explain the GPU’s threading model in more detail in Section 2), and *tid* is a numeric thread id. Memory accesses that involve each thread accessing the same address (Example 1) or consecutive threads accessing consecutive addresses (Example 2) fall within a single cache line, and so are considered coalesced. Memory accesses that have consecutive threads accessing non-consecutive addresses, as in Example 3, result in significant slowdowns: Example 3 will run about 8x slower than the other examples.

Discovering uncoalesced accesses statically introduces many challenges. Expressions used for array indexing are often complex, and their coalesced or uncoalesced nature must be propagated through arithmetic operations, to each use. The size of the data types involved in a memory access affects coalescing. The number of threads that actually execute a particular access affects coalescing as well; *e.g.*, Example 3 in Figure 1 is coalesced if only a single active thread reaches this statement. In this paper, we define GPUDrano, a simple but effective abstraction for detecting uncoalesced accesses that uses intra-procedural dataflow analysis to identify uncoalesced memory accesses in GPU programs statically. We target GPU code written for Nvidia’s CUDA programming model. GPUDrano makes the following contributions:

- To the best of our knowledge, GPUDrano is the first scalable static analysis for uncoalesced global memory accesses in GPU programs.
- We provide a formal definition of both our analysis and the memory coalescing bugs we wish to detect.
- GPUDrano leverages well-established program analysis techniques to improve scalability and is able to analyze thousand-line CUDA programs in seconds, while incorporating relevant information such as accounting for the set of active threads to reduce the number of false positives.
- We demonstrate that GPUDrano works in practice by implementing it in LLVM and detecting over a hundred real uncoalesced accesses in the well-

```

// t, N ↦ 0
if(tid+t+1 >= N) return;
x = tid+t+1; // x ↦ 1
for(y=t; y<N; y++){ // y ↦ 0
  xt = N*x + t; // xt ↦ T
  xy = N*x + y; // xy ↦ T
  ty = N*t + y; // ty ↦ 0
  A[xy] -= M[xt]*A[ty];
  if(y == t)
    B[x] -= M[xt]*B[t];
}

```

(a) Original Fan2 snippet

```

// t, N ↦ 0
if(tid+t >= N) return;
y = tid+t; // y ↦ 1
for(x=t+1; x<N; x++){ // x ↦ 0
  xt = N*x + t; // xt ↦ 0
  xy = N*x + y; // xy ↦ 1
  ty = N*t + y; // ty ↦ 1
  A[xy] -= M[xt]*A[ty];
  if(y == t)
    B[x] -= M[xt]*B[t];
}

```

(b) Fixed Fan2 snippet

Fig. 2: Kernel snippets from Gaussian Elimination program.

established Rodinia benchmark suite. We also validate GPUDrano against a dynamic analysis to show that GPUDrano has few or no false positives on most programs.

The remainder of this paper is organized as follows. [Section 2](#) describes the CUDA programming model and a real memory coalescing bug from our benchmarks. [Section 3](#) presents our formalization of CUDA programs, their executions, and uncoalesced accesses. [Section 4](#) describes the GPUDrano static analysis. [Section 5](#) describes the dynamic analysis we use to validate our GPUDrano implementation. [Section 6](#) discusses our experimental results, and [Section 7](#) related work. Finally, [Section 8](#) concludes.

2 Illustrative Example

We use an example GPU program to briefly illustrate the GPU programming model and the problem of uncoalesced accesses. GPUs follow an SIMT (Single Instruction Multiple Thread) execution model, where multiple threads execute the same sequence of instructions, often called a *kernel*. [Figure 2a](#) shows one such kernel, Fan2, from Gaussian Elimination program in Rodinia benchmark suite [\[5\]](#). The comments in the kernel can be ignored for now. The kernel performs row operations on matrix A (size $N \times N$) and vector B (size $N \times 1$) using the t^{th} column of a multiplier matrix M (size $N \times N$) and the t^{th} row of A and B . The kernel is a sequential procedure that takes in a thread id, tid , to distinguish executions of different threads. The kernel is executed for threads with ids in range $[0, N - t - 2]$. Each thread is assigned a distinct row and updates row $(\text{tid} + t + 1)$ of matrix A and vector B . Note that A , B and M reside in global memory and are shared across threads, while the remaining variables are private to each thread.

The GPU executes threads in bundles, or *warps*, where threads in each warp consist of consecutive ids and execute instructions in lock-step. The above kernel,

$S ::=$	statement
AS	assignment
if $\langle test \rangle$ then S_1 else S_2	conditional
while $\langle test \rangle$ do S	loop
$S_1; S_2$	sequence

Fig. 3: The grammar for kernel K .

for example, might be executed for warps w_0 with ids $[0, 31]$, w_1 with ids $[32, 63]$, and so on \dots . When a warp, say w_0 , accesses A using index xy in $\mathbf{A}[xy]$ for some iteration of y , the elements $A[N(t+1)+y]$, $A[N(t+2)+y]$, \dots , $A[N(t+32)+y]$ are fetched simultaneously. The elements are at least N locations apart from each other, and thus, separate transactions are required to access each element, which takes significant time and energy. This is an *uncoalesced* access. Access to $\mathbf{M}[xt]$ is similarly uncoalesced. Now, Figure 2b shows a fixed version of the kernel, where each thread is mapped to a column of the matrices A and M , instead of a row. The access to $\mathbf{A}[xy]$ by warp w_0 results in elements $A[Nx+t]$, $A[Nx+t+1]$, \dots , $A[Nx+t+31]$ to be accessed. These are consecutive elements, and thus, can be accessed in a single transaction. Access to $\mathbf{M}[xt]$ is similarly coalesced, and our experiments show a 25% reduction in run-time for the fixed kernel, when run for inputs with $N = 1024$.

3 Formalization of Uncoalesced Accesses

This section describes the GPU programming model and uncoalesced accesses formally. A GPU program is a tuple $\langle T, V_L, V_G, K \rangle$, where T represents the set of all threads; V_L and V_G represent the sets of variables residing in local and global memories respectively; and K represents the kernel or the sequence of instructions executed by the threads. The kernel K is defined by the grammar in Figure 3, and consists of assignments, conditionals and loops. The set V_L further contains a special read-only variable, `tid`, initialized with the thread id of the thread. The variable can appear in the right-hand-side of assignments and helps distinguish executions of different threads.

We next present a simple operational semantics for GPU programs. We use a simplified execution model, where *all* threads in the program execute instructions in lock-step. While the standard GPU execution model is more flexible, this assumption simplifies the semantics without affecting the detection of memory coalescing bugs, and has been used in a previous formalization [4].

Excluded GPU features. The GPU programming model represents threads in a two-level hierarchy, where a bunch of threads form a *thread-block* and the thread-blocks together form the set of all threads. Further, threads have access to a block-level memory space, *shared memory*, used to share data between threads within a block. Lastly, threads within a block can synchronize on a `_syncthreads()` barrier. These features do not directly affect uncoalesced accesses and have been excluded here for the ease of presentation.

3.1 Semantics

To describe the semantics, we define two entities: the state σ and the active set of threads π . The state σ maps variables to a type-consistent value ν . It consists of a copy of local variables per thread and a copy of global variables, and thus, is a function $(V_L \times T) \cup V_G \rightarrow \mathcal{V}$. We further use \perp to represent an undefined or error state. The active set of threads π is a subset of T and defines a set of threads for which the execution is active. Now, the semantics for a statement S are given by the function $\llbracket S \rrbracket$, where $\llbracket S \rrbracket(\sigma, \pi) = \sigma'$ represents the execution of statement S in state σ for threads in set π to generate a new state σ' .

Assignments. We first define the semantics for assignment statements when executed by a single thread t i.e. $\llbracket AS \rrbracket(\sigma, t) = \sigma'$. Let $l \in V_L$ and $g \in V_G$ represent a local and global variable, respectively. Let v represent a generic variable. Further, let the variables be either scalars or arrays. An assignment is of the form $[E := e]$, where E is the expression being updated, and consists of either a scalar variable v or an array variable indexed by a local $v(l)$; and e is an expression whose value is assigned to E , and is built using scalar variables, array variables indexed by locals, constants, and arithmetic and boolean operations on them. Note that in an assignment at least one of E and e must be a local scalar variable l . We distinguish two types of assignments: *global array read* $[l' := g(l)]$, where the global array g indexed by l is read into l' , i.e. $\sigma'(l', t) = \sigma(g)(\sigma(l, t))$ and $\sigma'(v, t) = \sigma(v, t)$ for all $v \neq l$; and *global array write* $[g(l) := l']$, where g indexed by l is written with value of l' , i.e. $\sigma'(g)(\sigma(l, t)) = \sigma(l', t)$, $\sigma'(g)(v) = \sigma(g)(v)$ for all $v \neq \sigma(l, t)$, and $\sigma'(v, t) = \sigma(v, t)$ for all $v \neq g$.

We now define the semantics when an assignment is executed by a set of threads π , i.e. $\llbracket AS \rrbracket(\sigma, \pi) = \sigma'$. When the set π is empty, the state remains unchanged i.e. $\llbracket AS \rrbracket(\sigma, \phi) = \sigma$. When π is non-empty i.e. $\pi = \{t\} \cup \pi'$, the desired update is obtained by first executing AS for thread t , and then the other threads in π' . Thus, $\llbracket AS \rrbracket(\sigma, \pi) = \llbracket AS \rrbracket(\llbracket AS \rrbracket(\sigma, t), \pi')$. Note that, if different threads write to the same memory location, the execution is not deterministic and the updated state is set to the undefined state, i.e. $\llbracket AS \rrbracket(\sigma, t) = \perp$.

Sequences. The execution of sequence of statements $(S_1; S_2)$ is described by first executing S_1 , followed by S_2 i.e. $\llbracket S_1; S_2 \rrbracket(\sigma, \pi) = \llbracket S_2 \rrbracket(\llbracket S_1 \rrbracket(\sigma, \pi), \pi)$.

Conditionals. Next consider $\llbracket \text{if } l \text{ then } S_1 \text{ else } S_2 \rrbracket(\sigma, \pi) = \sigma'$, where $\langle test \rangle$ consists of a local boolean variable l . The semantics *serializes* the execution of statements S_1 and S_2 . Let the set of threads for which the predicate $\sigma(l, t)$ is **true** be π_1 . The threads in π_1 first execute S_1 to get the state σ_1 i.e. $\llbracket S_1 \rrbracket(\sigma, \pi_1) = \sigma_1$. Next, the remaining threads execute S_2 in state σ_1 to get the final updated state i.e. $\llbracket S_2 \rrbracket(\sigma_1, \pi \setminus \pi_1) = \sigma'$. Note that, similar to assignments, if the same location is read or written by a thread executing the **if** branch and another thread executing the **else** branch with one of the accesses being writes, there is a potential conflict between the two accesses, and the final state σ' is set to \perp .

Loops. We next describe the semantics for loops $\llbracket \text{while } l \text{ do } S \rrbracket(\sigma, \pi) = \sigma'$. We first consider semantics for terminating loops. The loop execution terminates when there are no threads active in the loop, and it is repeated until then. Formally, if there exist $\sigma_1, \pi_1, \sigma_2, \pi_2, \dots, \sigma_k, \pi_k$, such that σ_i and π_i represent

the state and the active set of threads at the beginning of the i th iteration of the loop, i.e. $\sigma_1 = \sigma$, $\pi_1 = \{t \in \pi : \sigma(l, t) = \mathbf{true}\}$, $\sigma_{i+1} = \llbracket S \rrbracket(\sigma_i, \pi_i)$, and $\pi_{i+1} = \{t \in \pi_i : \sigma_{i+1}(l, t) = \mathbf{true}\}$, and the last active set is empty, $\pi_k = \phi$, then $\sigma' = \sigma_k$. If the loop is non-terminating, σ' is assigned the undefined state \perp .

Reachable Configurations. We now define the set \mathcal{R} of configurations reachable during a kernel’s execution. A configuration is a tuple (σ, π, S) , where σ is the current state, π is the current active set of threads, and S is the next statement to be executed. We give an inductive definition for \mathcal{R} . The initial configuration (σ_0, T, K) belongs to \mathcal{R} , where σ_0 is the initial state, T is the set of all threads and K is the kernel. In the recursive case, suppose (σ, π, S) belongs to \mathcal{R} . When $S = S_1; S_2$, the configuration (σ, π, S_1) belongs to \mathcal{R} , since S_1 is the next statement to be executed. Further, if the state after executing S_1 , $\sigma' = \llbracket S_1 \rrbracket(\sigma, \pi)$, is not undefined i.e. $\sigma' \neq \perp$, then (σ', π, S_2) also belongs to \mathcal{R} . Similarly, when S is a conditional **[if l then S_1 else S_2]**, both **if** and **else** branches are reachable, and thus, (σ, π_1, S_1) and (σ, π_2, S_2) belong to \mathcal{R} , where $\pi_1 = \{t \in \pi : \sigma(l, t) = \mathbf{true}\}$ and $\pi_2 = \pi \setminus \pi_1$. Lastly, when S is a loop **[while l do S']**, the configuration (σ, π', S') , where $\pi' = \{t \in \pi : \sigma(l, t) = \mathbf{true}\}$, is reachable. Further, if the state after executing S' is not undefined, the configuration $(\llbracket S' \rrbracket(\sigma, \pi'), \pi', S)$ is also reachable.

3.2 Uncoalesced Global Memory Accesses

To define uncoalesced global memory accesses, we first describe how the global memory is accessed by a GPU. Let the memory bandwidth for global memory be η bytes i.e. the GPU can access η contiguous bytes from the memory in one transaction. When a warp of threads with consecutive thread indices W issues a read or write to the global memory, the addresses accessed by the active threads are coalesced together into as few transactions as possible. If the number of transactions is above a threshold τ , there is an uncoalesced access.

We now define uncoalesced accesses formally. Consider the configuration (σ, π, AS) , where AS is a global array read $[l' := g(l)]$ or a global array write $[g(l) := l']$ and g is a global array with each element of size k bytes. Let W be a warp of threads with consecutive thread indices. Let the addresses accessed by the warp, $\Gamma(\sigma, \pi, AS, W)$, be defined as,

$$\Gamma(\sigma, \pi, AS, W) = \bigcup_{t \in W \cap \pi} [\sigma(l, t).k, \sigma(l, t).k + k - 1]$$

Now, each contiguous set of η bytes is accessed in one transaction. Thus, the number of transactions $N(\sigma, \pi, AS, W)$ required for the access equals the number of unique elements in the set $\{\lfloor a/\eta \rfloor : a \in \Gamma(\sigma, \pi, AS, W)\}$. If $N(\sigma, \pi, AS, W)$ is greater than threshold τ for some warp W , the configuration (σ, π, AS) is an “uncoalesced” configuration. A global array access AS is uncoalesced, if an uncoalesced configuration involving the access is reachable.

For most current GPUs, the bandwidth $\eta = 128$ bytes, and warp size $|W| = 32$. We use the threshold $\tau = 1$, so that accesses that require more than one

transaction are flagged as uncoalesced. Suppose the index variable l in a global array access is a linear function of tid , i.e. $l \equiv c \cdot \text{tid} + c_0$. The range of addresses accessed by a completely active warp W is $(31|kc| + k - 1)$ bytes and thus, the number of transactions N required is at least $|kc|/4$. If $k \geq 4$ bytes and $|c| \geq 1$ (with one of the inequalities being strict), N is greater than 1, and hence, the access is uncoalesced. We refer to such uncoalescing, where the range of addresses accessed by a warp is large, as *range-based* uncoalescing.

Alternately, an uncoalesced access can occur due to alignment issues, where the range of accessed locations is small but mis-aligned with the cache block boundaries. Suppose $k = 4$ and $c = 1$, but $c_0 = 8$. The addresses accessed by a warp W with tids $[0, 31]$ are $[32, 159]$ and require two transactions, even though the range of locations is 127 bytes which is less than the bandwidth. We refer to such accesses as *alignment-based* uncoalesced accesses.

4 Static Analysis

This section presents a static compile-time analysis to identify uncoalesced accesses. We use abstract interpretation [8,17] for the analysis, where-in we first define an abstraction of the state and the active set of threads. The abstraction captures features of the kernel execution essential to identify uncoalesced accesses. It tracks values, particularly access indices, as a function of tid , and for the indices with potentially large linear or non-linear dependence on tid , the analysis flags the corresponding global array access as uncoalesced. Further, if a segment of code is executed only by a single thread (which is often the case when some sequential work needs to be done), a single transaction is required for an access and it cannot be uncoalesced. Hence, our abstraction also tracks whether single or multiple threads are active during the execution of a statement.

After defining the abstraction, we associate abstract semantics with the statements in kernels, computable at compile-time, that preserve the abstraction. We then present our algorithm to execute kernels abstractly and identify global array accesses which can potentially be uncoalesced. Finally, we describe our implementation for the analysis.

Example. Before diving into the details of the analysis, let’s consider the example in Figure 2a. Our abstraction tracks local variables as a function of tid . All variables that are independent of tid are assigned value 0 in the abstraction. Thus, variables t and N are assigned the value 0 initially (shown in comments). Further, variables y and ty are constructed from tid -independent variables, and hence, assigned 0. Next, all variables that are linear function of tid with coefficient 1 (i.e. of the form $\text{tid} + c$), are assigned value 1. The variable x is therefore assigned 1. Lastly, all variables that are either non-linear function of tid or linear function with possibly greater than one coefficient are assigned \top . Variable xt , for example, is assigned the expression $N(\text{tid} + t + 1) + t$, where the coefficient for tid is N . Since N can be greater than one, xt is assigned \top . Similarly, variable xy is assigned \top . Now, global array accesses where the index variable has value \top , are flagged as uncoalesced. Hence, accesses $\mathbf{A}[xy]$ and $\mathbf{M}[xt]$ are flagged

as uncoalesced. Note that in the fixed kernel in Figure 2b, none of the index variables are \top , and hence, none of the accesses are flagged as uncoalesced.

4.1 Abstraction

We now formally define our abstraction. Let $\alpha()$ be the abstraction function. The abstraction of state $\hat{\sigma}$ only tracks values of local scalar variables. We observe that indirect indexing through arrays is rare for coalesced accesses, and hence we conservatively flag all such accesses as uncoalesced. Further, we use a different abstraction for integer and boolean variables. We assign a single abstract value to each local variable, that tracks its dependency on tid . For integer variables, we use the set $\widehat{\mathcal{V}}_{int} = \{\perp, 0, 1, -1, \top\}$ to abstract values. The value \perp represents undefined values, while \top represents all values. The remaining values are defined here. Let l be a local variable.

$$\alpha(\sigma)(l) = \begin{cases} 0, & \text{exists } c_0 \text{ s.t. for all } t \in T, \sigma(l, t) = c_0 \\ 1, & \text{exists } c_0 \text{ s.t. for all } t \in T, \sigma(l, t) = \text{tid}(t) + c_0 \\ -1, & \text{exists } c_0 \text{ s.t. for all } t \in T, \sigma(l, t) = -\text{tid}(t) + c_0 \end{cases}$$

i.e. the abstract value 0 represents values constant across threads; 1 represents values that are a linear function of tid with coefficient 1; and, -1 represents values that are a linear function with coefficient -1. This abstraction is necessary to track dependency of access indices on tid .

We use the set $\widehat{\mathcal{V}}_{bool} = \{\perp, \top, \top^-, \text{F}, \text{F}^-, \text{TF}, \text{TT}^-, \text{FF}^-, \top\}$ to abstract boolean variables. Again \perp and \top represent the undefined value and all values, respectively. The remaining values are defined here.

$$\alpha(\sigma)(l) \equiv \begin{cases} \top, & \text{for all } t \in T, \sigma(l, t) = \text{true} \\ \top^-, & \text{exists } t \in T \text{ s.t. } \sigma(l, t) = \text{false} \\ & \text{and for all } t' \in T \setminus t, \sigma(l, t') = \text{true} \\ \text{F}, & \text{for all } t \in T, \sigma(l, t) = \text{false} \\ \text{F}^-, & \text{exists } t \in T \text{ s.t. } \sigma(l, t) = \text{true} \\ & \text{and for all } t' \in T \setminus t, \sigma(l, t') = \text{false} \end{cases}$$

i.e. the abstract value \top represents values **true** for all threads; \top^- represents values **true** for all but one thread; **F** represents values **false** for all threads; F^- represents values **false** for all but one thread. Further, we construct three additional boolean values: $\text{TF} = \{\top, \text{F}\}$ representing values **true** or **false** for all threads, $\text{TT}^- = \{\top, \top^-\}$ representing values **false** for at most one thread, and $\text{FF}^- = \{\text{F}, \text{F}^-\}$ representing values **true** for at most one thread. We only use these compound values in our analysis, along with \perp and \top . We use them to abstract branch predicates in kernels. This completes the abstraction for state. Note that $\hat{\sigma}$ is function $V_L \rightarrow \widehat{\mathcal{V}}_{int} \cup \widehat{\mathcal{V}}_{bool}$.

Now, the active set of threads π can be seen as a predicate on the set of threads T . We observe that if at most one thread is active for a global array access, a single transaction is required to complete the access and hence, it is always coalesced. Thus, in our abstraction for π , we only track if it consists of

at most one thread or an arbitrary number of threads. These can be abstracted by boolean values FF^- and \top respectively, and thus, $\hat{\pi} \in \{\text{FF}^-, \top\}$.

Lastly, our abstraction for boolean and integer variables induces a natural complete lattice on sets $\widehat{\mathcal{V}}_{int}$ and $\widehat{\mathcal{V}}_{bool}$. These lattices can be easily extended to complete lattices for the abstract states and active sets of threads.

Justification. We designed our abstraction by studying the benchmark programs in Rodinia. We have already motivated the abstract values 0, 1 and \top for integer variables in the example above. We found coefficient -1 for `tid` in a few array indices, which led to the abstract value -1. There were also instances where values 1 and -1 were added together to generate 0 or `tid`-independent values. Next, the values FF^- and TT^- were motivated by the need to capture predicates in conditionals where one of the branches consisted of at most one active thread. Lastly, the value TF was necessary to distinguish conditionals with `tid`-dependent and `tid`-independent predicates.

4.2 Abstract Semantics

We briefly describe the abstract semantics $\llbracket \widehat{S} \rrbracket(\hat{\sigma}, \hat{\pi}) = \hat{\sigma}'$ for a statement S , which is the execution of S in an abstract state $\hat{\sigma}$ for abstract active set $\hat{\pi}$ to generate the abstract state $\hat{\sigma}'$. We first consider abstract computation of values of local expressions e (involving only local scalar variables) in state $\hat{\sigma}$, $\llbracket \widehat{e} \rrbracket(\hat{\sigma})$. Local scalar variable l evaluates to its value in $\hat{\sigma}$, $\hat{\sigma}(l)$. Constants evaluate to the abstract value 0 (TF if boolean). Index `tid` evaluates to 1. Arithmetic operations on abstract values are defined just as regular arithmetic, except all values that do not have linear dependency on `tid` with coefficient 0, 1 or -1, are assigned \top . For example, $[1 + 1] = \top$ since the resultant value has a dependency of 2 on `tid`. Boolean values are constructed from comparison between arithmetic values. Equalities $[\hat{\nu}_1 = \hat{\nu}_2]$ are assigned a boolean value FF^- , and inequalities $[\hat{\nu}_1 \neq \hat{\nu}_2]$ a boolean value TT^- , where one of ν_1 and ν_2 equals 1 or -1, and the other 0. Note that this is consistent with our abstraction. The equalities are of the form $[\text{tid} = c]$, for some constant c , and are true for at most one thread. The inequalities are of the form $[\text{tid} \neq c]$ and are true for all except one thread. For boolean operations, we observe that $\neg \text{TT}^- = \text{FF}^-$, $[\text{FF}^- \wedge b] = \text{FF}^-$, and $[\text{TT}^- \vee b] = \text{TT}^-$, for all $b \in \{\text{TF}, \text{FF}^-, \text{TT}^-, \top\}$. Other comparison and boolean operations are defined similarly.

We next define the abstract semantics for different types of assignments AS in a state $\hat{\sigma}$, $\llbracket \widehat{AS} \rrbracket(\hat{\sigma}) = \hat{\sigma}'$. For local assignments $[l := e]$ where e is a local expression, l is updated with value of expression e , $\llbracket \widehat{e} \rrbracket(\hat{\sigma})$. For reads $[l := g]$, where g is a global scalar variable, all threads receive the same value, and the new value is `tid`-independent. Hence, l is updated to 0 (TF if boolean). For array reads $[l := v(l')]$, where $\hat{\sigma}(l') = 0$, all threads access the same element in the array v , and receive the same value. Thus, the updated value is again 0 (TF if boolean). Lastly for array reads where $\hat{\sigma}(l') \neq 0$, the read could return values that are arbitrary function of `tid` (since we do not track the values for arrays), and hence, the updated value is \top .

$$\begin{array}{c}
\frac{\widehat{\llbracket AS \rrbracket}(\hat{\sigma}) = \hat{\sigma}'}{\widehat{\llbracket AS \rrbracket}(\hat{\sigma}, \hat{\pi}) = \hat{\sigma}'} \quad \text{ASSIGN} \\
\hat{\pi}_1 = [\hat{\pi} \wedge \hat{\sigma}(l)] \\
\hat{\pi}_2 = [\hat{\pi} \wedge \neg \hat{\sigma}(l)] \\
\widehat{\llbracket S_1 \rrbracket}(\hat{\sigma}, \hat{\pi}_1) = \hat{\sigma}_1 \\
\widehat{\llbracket S_2 \rrbracket}(\hat{\sigma}, \hat{\pi}_2) = \hat{\sigma}_2 \\
\hat{\sigma}_3 = \Phi_{\{S_1, S_2\}}^{\hat{\sigma}(l)}(\hat{\sigma}_1, \hat{\sigma}_2) \\
\hline
\widehat{\llbracket \text{if } l \text{ then } S_1 \text{ else } S_2 \rrbracket}(\hat{\sigma}, \hat{\pi}) = \hat{\sigma}_3 \quad \text{ITE}
\end{array}
\qquad
\begin{array}{c}
\frac{\widehat{\llbracket S_1 \rrbracket}(\hat{\sigma}, \hat{\pi}) = \hat{\sigma}_1 \quad \widehat{\llbracket S_2 \rrbracket}(\hat{\sigma}_1, \hat{\pi}) = \hat{\sigma}_2}{\widehat{\llbracket S_1; S_2 \rrbracket}(\hat{\sigma}, \hat{\pi}) = \hat{\sigma}_2} \quad \text{SEQ} \\
\hat{\pi}' = [\hat{\pi} \wedge \hat{\sigma}(l)] \\
\widehat{\llbracket S \rrbracket}(\hat{\sigma}, \hat{\pi}') = \hat{\sigma}_1 \\
\hat{\sigma}_2 = \Phi_{\{\text{skip}, S\}}^{\hat{\sigma}(l)}(\hat{\sigma}, \hat{\sigma}_1) \\
\widehat{\llbracket \text{while } l \text{ do } S \rrbracket}(\hat{\sigma}_2, \hat{\pi}') = \hat{\sigma}_3 \\
\hline
\widehat{\llbracket \text{while } l \text{ do } S \rrbracket}(\hat{\sigma}, \hat{\pi}) = \hat{\sigma}_3 \quad \text{WHILE}
\end{array}$$

Fig. 4: Abstract semantics for compound statements.

We now define abstract semantics for the compound statements. We use rules in Figure 4 to describe them formally. Note that, our abstract semantics for assignments are oblivious to the set of threads $\hat{\pi}$, and thus, the [ASSIGN] rule extends these semantics to an arbitrary set of threads. The [SEQ] rule similarly extends the semantics to sequence of statements. The [ITE] rule describes the semantics for conditionals. The sets $\hat{\pi}_1$ and $\hat{\pi}_2$ represent the new active set of threads for the execution of S_1 and S_2 . Note that $\hat{\pi}_1 = [\hat{\pi} \wedge \hat{\sigma}(l)]$, and gets a value FF^- , only if either $\hat{\pi}$ or $\hat{\sigma}(l)$ is FF^- . The new set of threads π_1 has at most one thread, only if either the incoming set π or the predicate $\sigma(l, t)$ is true for at most one thread. Hence, $\hat{\pi}_1$ correctly abstracts the new set of the threads for which S_1 is executed. A similar argument follows for $\hat{\pi}_2$. Now, the concrete value for predicate $\sigma(l, t)$ is not known at compile time, and a thread could execute either of S_1 or S_2 . Hence, our abstract semantics executes both, and merges the two resulting states to get the final state, i.e. $\Phi_{\{S_1, S_2\}}^{\hat{\sigma}(l)}(\hat{\sigma}_1, \hat{\sigma}_2)$.

The merge operation is a non-trivial operation and depends on the branch predicate $\hat{\sigma}(l)$. If $\hat{\sigma}(l)$ is TF or tid -independent, all threads execute either the **if** branch or the **else** branch, and final value of a variable l is one of the values $\hat{\sigma}_1(l)$ and $\hat{\sigma}_2(l)$. In the merged state, our semantics assigns it a merged value $\hat{\sigma}_1(l) \sqcup \hat{\sigma}_2(l)$ or the join of the two values, a value that subsumes both these values. When $\hat{\sigma}(l)$ is tid -dependent, however, this merged value does not suffice. Consider, for example, $y := (\text{tid} < N)? 10 : 20$. While on both the branches, y is assigned a constant (abstract value 0), the final value is a non-linear function of tid (abstract value \top), even though the join of the two values is 0. Hence, in such cases, when the predicate is tid -dependent and the variable l is assigned a value in S_1 or S_2 , the merged value is set to \top .

The [WHILE] rule describes the abstract semantics for loops. Note that similar to conditionals, it is not known whether a thread executes S or not. Thus, the rule first transforms the original state $\hat{\sigma}$ into the merge of $\hat{\sigma}$ and the execution of S on $\hat{\sigma}$ and repeats this operation, until the fixed point is reached and the state does not change on repeating the operation. Note that our abstract semantics for different statements are monotonic. The merge operation Φ is also monotonic.

The abstract state can have only finite configurations, since each variable gets a finite abstract value. Thus, the fixpoint computation always terminates.

4.3 Detecting Uncoalesced Accesses

We first define the set of abstract configurations that are reachable during the abstract execution of the kernel. An abstract configuration is the tuple $(\hat{\sigma}, \hat{\pi}, S)$. The initial abstract configuration is $(\alpha(\sigma_0), \top, K)$, and is reachable. The other abstract reachable configurations can be defined by a similar recursive definition as that for reachable configurations.

Now, an abstract configuration $(\hat{\sigma}, \hat{\pi}, AS)$ is “uncoalesced”, where AS is a global array read $[l' := g(l)]$ or global array write $[g(l) := l']$ and g is a global array with elements of size k , if both these conditions hold:

- $\hat{\pi} = \top$ i.e. the access is potentially executed by more than one thread.
- $(\hat{\sigma}(l) = \top) \vee (\hat{\sigma}(l) \in \{1, -1\} \wedge k > 4)$ i.e. l is a large linear or non-linear function of tid, or it is a linear function of tid with unit coefficient and the size of elements of array g is greater than 4 bytes.

The analysis computes the set of abstract reachable configurations by executing the kernel using the abstract semantics, starting from the abstract initial configuration. It reports a global array access AS as uncoalesced, if an abstract uncoalesced configuration involving AS is reached during the abstract execution of the kernel.

Correctness. We show that for all global array accesses AS , if a range-based uncoalesced configuration involving a global array access AS is reachable, the analysis identifies it as uncoalesced. We first note that our abstract semantics preserve the abstraction. Hence, for any reachable configuration (σ, π, AS) , there exists an abstract reachable configuration $(\hat{\sigma}, \hat{\pi}, AS)$ that is an overapproximation of its abstraction i.e. $\alpha(\sigma) \sqsubseteq \hat{\sigma}$ and $\alpha(\pi) \sqsubseteq \hat{\pi}$. Now, for a range-based uncoalesced configuration to occur, the access needs to be executed by more than one thread, and thus $\alpha(\pi) = \top$. Further, the access index l either has non-linear dependence on tid, in which case $\alpha(\sigma)(l) = \top$, or as noted in Section 3.2, the index has linear dependence with one of $k > 4$ or $|c| > 1$, which again leads to an abstract uncoalesced configuration. Hence, GPUDrano identifies all range-based uncoalesced configurations as uncoalesced. There are no guarantees for alignment-based uncoalescing, however. This gives some evidence for the correctness of the analysis.

4.4 Implementation

We have implemented the analysis in the gpucc CUDA compiler [23], an open-source compiler based on LLVM. We implement the abstract semantics defined above. We work with an unstructured control flow graph representation of the kernel, where conditionals and loops are not exposed as separate units. So, we simplify the semantics at the cost of being more imprecise. We implement the

abstract computation of local expressions and the abstract semantics for assignments exactly. We however differ in our implementation for the merge operation. Consider a conditional statement `[if l then S_1 else S_2]`. Let the states after executing S_1 and S_2 be $\hat{\sigma}_1$ and $\hat{\sigma}_2$. The merge of states $\Phi_{\{S_1, S_2\}}^{\hat{\sigma}^{(l)}}(\hat{\sigma}_1, \hat{\sigma}_2)$ after the conditional is contingent on `tid`-dependence of the value of l at the beginning of the conditional. This information requires path-sensitivity and is not available in the control flow graph at the merge point. Therefore, we conservatively assume l to be `tid`-dependent. We use the SSA representation of control flow graph, where variables assigned different values along paths S_1 and S_2 are merged in special `phi` instructions after the conditional. We conservatively set the merged value to \top for such variables. The values of remaining variables remain unchanged. This completes the implementation of merge operation. We define the set of active threads $\hat{\pi}'$ after the conditional as $\hat{\pi}_1 \sqcup \hat{\pi}_2$, or the join of incoming active sets from S_1 and S_2 . The new active set $\hat{\pi}'$ equals the active set $\hat{\pi}$ before the conditional. If $\hat{\pi} = \top$, it must be split into $\hat{\pi}_1$ and $\hat{\pi}_2$ such that at least one of the values is \top and hence, $\hat{\pi}' = \top$. Similarly, when $\hat{\pi} = \text{FF}^-$, both $\hat{\pi}_1$ and $\hat{\pi}_2$ equal FF^- , and hence, $\hat{\pi}' = \text{FF}^-$.

Limitations. Our implementation does not do a precise analysis of function calls and pointers, which are both supported by CUDA. In the implementation, we assume that call-context of a kernel is always empty, and function calls inside a kernel can have arbitrary side-effects and return values. We support pointer dereferencing by tracking two abstract values for each pointer variable, one for the address stored in the pointer and the other for the value at the address. We do not implement any alias analyses, since we observe that array indices rarely have aliases. Our evaluation demonstrates that, despite these limitations, our static analysis is able to identify a large number of uncoalesced accesses in practice.

5 Dynamic Analysis

To gauge the accuracy of our GPUDrano static analysis, we have implemented a dynamic analysis for uncoalesced accesses. Being a dynamic analysis, it has full visibility into the memory addresses being accessed by each thread, as well as the set of active threads. Thus, the dynamic analysis can perfectly distinguish coalesced from uncoalesced accesses (for a given input). We use this to determine 1) whether the static analysis has missed any uncoalesced accesses and 2) how many of the statically-identified uncoalesced accesses are false positives.

The dynamic analysis is implemented as a pass in the `gpucc` CUDA compiler [23], which is based on LLVM. By operating on the LLVM intermediate representation (a type of high-level typed assembly), we can readily identify the instructions that access memory. So for every load and store in the program we insert instrumentation to perform the algorithm described below. A single IR instruction may be called multiple times in a program due to loops or recursion, so every store and load instruction is assigned a unique identifier (analogous to a program counter).

For every global memory access at runtime, we collect the address being accessed by each thread. Within each warp, the active thread with the lowest id is selected as the “computing thread” and it performs the bulk of the analysis. All active threads pass their addresses to the computing thread. The computing thread places all addresses into an array. This array will be at most of length n , where n is the warp size (if there are inactive threads, the size may be smaller). Next the computing thread determines all bytes that will be accessed by the warp, taking the size of the memory access into account. Note that, due to the SIMT programming model, the access size is the same for all threads in the warp since all threads execute the same instruction. Each byte accessed is divided by the size of the cache line using machine integer division. (for current generation Nvidia GPUs this number is 128 [18, §5.3.2]). Conceptually this assigns each address to a “bin” representing its corresponding cache line. For example, for a cache line of 128 bytes, $[0, 127] \mapsto 0$, $[128, 255] \mapsto 1$, etc. Finally, we count the number of unique bins, which is the total number of cache lines required. The computing thread prints the number of required cache lines, along with the assigned program counter, for post-processing.

A second, off-line step aggregates the information from each dynamic instance of an instruction by averaging. For example, if a load l executes twice, first touching 1 cache line and then touching 2 cache lines, the average for l will be 1.5 cache lines. If the average is 1.0 then l is coalesced, otherwise if the average is > 1.0 l is uncoalesced. The specific value of the average is sometimes useful, to distinguish accesses that are mildly uncoalesced (with averages just over 1.0), as we explore more in Section 6.

6 Evaluation

This section describes the evaluation of GPUDrano on the Rodinia benchmarks (version 3.1) [5]. Rodinia consists of GPU programs from various scientific domains. We run our static and dynamic analyses to identify existing uncoalesced accesses in these programs. We have implemented our analyses in LLVM version 3.9.0, and compile with `--cuda-gpu-arch=sm_30`. We use CUDA SDK version 7.5. We run our experiments on an Amazon EC2 instance with Amazon Linux 2016.03 (OS), an 8-core Intel Xeon E5-2670 CPU running at 2.60GHz, and an Nvidia GRID K520 GPU (Kepler architecture).

Table 1 shows results of our experiments. It shows the benchmark name, the lines of GPU source code analyzed, the manually-validated real uncoalesced accesses, and the number of uncoalesced accesses found and running time for each analysis. The Rodinia suite consists of 22 programs. We exclude 4 (hybridsort, kmeans, leukocyte, mummergpu) as they could not be compiled due to lack of support for texture functions in LLVM. We synonymously use “bugs” for uncoalesced accesses, though sometimes they are fundamental to the program and cannot be avoided. We next address different questions related to the evaluation.

Do uncoalesced accesses occur in real programs? We found 143 actual bugs in Rodinia benchmarks, with bugs in almost every program (Column “Real

Benchmark	LOC	Real-bugs	SA-bugs (real)	SA-runtime(s)	DA-bugs	DA-runtime(s)
backprop	110	7	0 (0)	0.14	7	5.23
bfs	35	7	7 (7)	0.07	0-7	3.89
b+tree	115	19	19 (19)	0.35	7	16.71
CFD	550	0	22 (0)	12.41	-	-
dwt2D	1380	0	16 (0)	5.99	n/a	3.72
gaussian	30	6	6 (6)	0.07	5-6	6.82
heartwall	1310	8	25 (8)	39.87	-	-
hotspot	115	3	2 (0)	0.75	3	0.89
hotspot3D	50	2	12 (2)	0.21	2	327.00
huffmann	395	21	26 (21)	0.68	3	2.42
lavaMD	180	9	9 (9)	0.73	5	511.60
lud	160	3	0 (0)	0.34	3	0.83
myocyte	3240	19	19 (19)	1,813.72	0	134.13
nn	10	4	4 (4)	0.06	2	0.13
nw	170	7	2 (2)	0.41	6	4.17
particle filter	70	4	3 (2)	0.58	4	11.62
pathfinder	80	3	0 (0)	0.22	3	4.25
srad_v1	275	2	14 (2)	0.33	2	185.00
srad_v2	250	9	0 (0)	1.38	9	53.94
streamcluster	45	10	10 (10)	0.11	-	-
		143	180 (111)		69	

Table 1: Results of GPUDrano’s static analysis (SA) and dynamic analysis (DA) on Rodinia benchmark programs. “-” indicates the DA hit the 2-hour timeout.

bugs” in Table 1). A few of the bugs involved random or irregular access to global arrays (bfs, particle filter). Such accesses are dynamic and data-dependent, and difficult to fix. Next, we found bugs where consecutive threads access rows of global matrices, instead of columns (gaussian). Such bugs could be fixed by assigning consecutive threads to consecutive columns or changing the layout of matrices, but this is possible only when consecutive columns can be accessed in parallel. Another common bug occurred when data was allocated as an array of structures instead of a structure of arrays (nn, streamcluster). A closely related bug was one where the array was divided into contiguous chunks and each chunk was assigned to a thread, instead of allocating elements in a round-robin fashion (myocyte, streamcluster). There were some bugs which involved reduction operations (for example, sum) on arrays (heartwall, huffmann). These bugs do not have a standard fix, and some of the above techniques could be applicable. A few bugs were caused by alignment issues where accesses by a warp did not align with cache-block boundaries, and hence, got spilled over to multiple blocks. These were caused, first, when the input matrix dimensions were not a multiple of the warp size which led consecutive rows to be mis-aligned (backprop, hotspot3D), or when the whole array was misaligned due to incorrect padding (b+tree). These could be fixed by proper padding.

Which real bugs does static analysis miss? While the static analysis catches a significant number of bugs (111 out of 143), it does miss some in practice. We found two primary reasons for this. 22 of the missed bugs depend on the second dimension of the `tid` vector, while we only considered the smallest dimension in our analysis. Uncoalesced accesses typically do not depend on higher dimensions unless the block dimensions are small or not a multiple of the warp size. We modified our analysis to track the second dimension and observed that all these bugs were caught by the static analysis, at the cost of 20 new false positives. Eight of the remaining missed bugs were alignment bugs which were caused by an unaligned offset added to `tid`. The actual offsets are challenging to track via static analysis. Two missed bugs (particle filter) were due to an implementation issue with conditionals which we will address in the future.

What false positives does static analysis report? For most programs, GPUDrano reports few or no false positives. The primary exceptions are CFD, `dwt2D` and `heartwall`, which account for the bulk of our false positives. A common case occurred when `tid` was divided by a constant, and multiplied back by the same constant to generate the access index (`heartwall`, `huffman`, `srad_v1`). Such an index should not lead to uncoalesced accesses. The static analysis, however, cannot assert that the two constants are equal, since we do not track exact values, and hence, sets the access index to \top , and reports any accesses involving the index as uncoalesced. Another type of false positive occurred when access indices were non-linear function of `tid`, but consecutive indices differed by at most one, and led to coalesced accesses. Such indices were often either generated by indirect indexing (CFD, `srad_v1`) or by assigning values in conditionals (`heartwall`, `hotspot`, `hotspot3D`). In both cases, our static analysis conservatively assumed them to be uncoalesced. Lastly, a few false positives happened because the access index was computed via a function call (`_mul24`) which returned a coalesced index (`huffmann`), though we conservatively set the index to \top .

How scalable is static analysis? As can be noted, the static analysis is quite fast, and finishes within seconds for most benchmarks. The largest benchmark, `myocyte`, is 3240 lines of GPU code, with the largest kernel containing 930 lines. The kernels in `myocyte` consist of many nested loops, and it appears the static analysis takes significant time computing fixed points for these loops.

How does static analysis compare with dynamic analysis? The dynamic analysis misses nearly half the bugs in our benchmarks. We found several benchmarks where different inputs varied the number of bugs reported (`bfs`, `gaussian`, `lud`). Similarly, the analysis finds bugs along a single execution path, so all bugs in unexecuted branches or uncalled kernels were not found. Due to compiler optimizations it can be difficult to map the results of dynamic analysis back to source code. In `dwt2D`, we were unable to do so due to multiple uses of C++ templates. Moreover, the dynamic analysis does not scale to long-running programs, as it incurs orders of magnitude of slowdown. While none of our benchmarks execute for more than 5 seconds natively, several did not finish with the dynamic analysis within our 2-hour limit (CFD, `heartwall`, `streamcluster`).

7 Related Work

While the performance problems that uncoalesced accesses cause are well understood [18, §5.3.2], there are few static analysis tools for identifying them.

Several compilers for improving GPU performance [24,11,7,20,21,3] incorporate some static analysis for uncoalesced global memory accesses, but these analyses are described informally and not evaluated for precision. Some of these systems also exhibit additional restrictions, such as CuMAPz’s [11] reliance on runtime traces, CUDA-lite’s [21] use of programmer annotations, or [20,3] which are applicable only to programs with affine access patterns. Some systems for optimizing GPU memory performance, like Dymaxion [6], eschew static analysis for programmer input instead. GPUDrano’s precision could likely help improve the performance of the code generated by these prior systems and reduce programmer effort. [1] describes in a short paper the preliminary implementation of CUPL, a static analysis for uncoalesced global memory accesses. While CUPL shares similar goals as GPUDrano, no formalization or detailed experimental results are described.

GKLEE [15] is a symbolic execution engine for CUDA programs. It can detect uncoalesced accesses to global memory (along with data races), but due to the limitations of its underlying SMT solver it cannot scale to larger kernels or large numbers of threads. The PUG verifier for GPU kernels [14] has also been extended to detect uncoalesced memory accesses [10], but PUG is less scalable than GKLEE. In contrast, GPUDrano’s static analysis can abstract away the number of threads actually used by a kernel.

[9] uses dynamic analysis to identify uncoalesced global memory accesses, and then uses this information to drive code transformations that produce coalesced accesses. GPUDrano’s static analysis is complementary, and would eliminate [9]’s need to be able to run the program on representative inputs.

There are many programming models that can generate code for GPUs, including proposals to translate legacy OpenMP code [12,13] or C code [3,2,22], and new programming models such as OpenACC [19] and C++ AMP [16]. An analysis such as GPUDrano’s could help improve performance in such systems, by identifying memory coalescing bottlenecks in the generated GPU code.

8 Conclusion

This paper presents GPUDrano, a scalable static analysis for uncoalesced global memory accesses in GPU programs. We formalize our analysis, and implement GPUDrano in LLVM. We apply GPUDrano to a range of GPU kernels from the Rodinia benchmark suite. We have evaluated GPUDrano’s accuracy by comparing it to a dynamic analysis that is fully precise for a given input, and found that the GPUDrano implementation is accurate in practice and reports few false positives for most programs. Fixing these issues can lead to performance improvements of up to 25% for the gaussian benchmark.

We would like to thank anonymous reviewers for their valuable feedback. This research was supported by NSF awards CCF-1138996 and XPS-1337174.

References

1. Amilkanthwar, M., Balachandran, S.: CUPL: A compile-time uncoalesced memory access pattern locator for CUDA. In: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing. pp. 459–460. ICS '13, ACM, New York, NY, USA (2013), <http://doi.acm.org/10.1145/2464996.2467288>
2. Baskaran, M.M., Bondhugula, U., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P.: Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 1–10. PPOPP '08, ACM, New York, NY, USA (2008), <http://doi.acm.org/10.1145/1345206.1345210>
3. Baskaran, M.M., Bondhugula, U., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P.: A compiler framework for optimization of affine loop nests for GPGPUs. In: Proceedings of the 22Nd Annual International Conference on Supercomputing. pp. 225–234. ICS '08, ACM, New York, NY, USA (2008), <http://doi.acm.org/10.1145/1375527.1375562>
4. Betts, A., Chong, N., Donaldson, A.F., Ketema, J., Qadeer, S., Thomson, P., Wickerson, J.: The design and implementation of a verification technique for GPU kernels. *ACM Trans. Program. Lang. Syst.* 37(3), 10:1–10:49 (May 2015), <http://doi.acm.org/10.1145/2743017>
5. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.H., Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing. In: 2009 IEEE International Symposium on Workload Characterization (IISWC). pp. 44–54 (Oct 2009)
6. Che, S., Sheaffer, J.W., Skadron, K.: Dymaxion: Optimizing memory access patterns for heterogeneous systems. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 13:1–13:11. SC '11, ACM, New York, NY, USA (2011), <http://doi.acm.org/10.1145/2063384.2063401>
7. Chen, G., Wu, B., Li, D., Shen, X.: Purple: An extensible optimizer for portable data placement on GPU. In: Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture. pp. 88–100. MICRO-47, IEEE Computer Society, Washington, DC, USA (2014), <http://dx.doi.org/10.1109/MICRO.2014.20>
8. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 238–252. POPL '77, ACM, New York, NY, USA (1977), <http://doi.acm.org/10.1145/512950.512973>
9. Fauzia, N., Pouchet, L.N., Sadayappan, P.: Characterizing and enhancing global memory data coalescing on GPUs. In: Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization. pp. 12–22. CGO '15, IEEE Computer Society, Washington, DC, USA (2015), <http://dl.acm.org/citation.cfm?id=2738600.2738603>
10. Jinpeng Lv, Guodong Li, Alan Humphrey, and Ganesh Gopalakrishnan: Performance degradation analysis of GPU kernels. In: Workshop on Exploiting Concurrency Efficiently and Correctly (2011)
11. Kim, Y., Shrivastava, A.: CuMAPz: A tool to analyze memory access patterns in CUDA. In: Proceedings of the 48th Design Automation Conference. pp. 128–

133. DAC '11, ACM, New York, NY, USA (2011), <http://doi.acm.org/10.1145/2024724.2024754>
12. Lee, S., Eigenmann, R.: OpenMPC: Extended OpenMP programming and tuning for GPUs. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–11. SC '10, IEEE Computer Society, Washington, DC, USA (2010), <https://doi.org/10.1109/SC.2010.36>
13. Lee, S., Min, S.J., Eigenmann, R.: OpenMP to GPGPU: A compiler framework for automatic translation and optimization. In: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 101–110. PPOPP '09, ACM, New York, NY, USA (2009), <http://doi.acm.org/10.1145/1504176.1504194>
14. Li, G., Gopalakrishnan, G.: Scalable SMT-based verification of GPU kernel functions. In: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 187–196. FSE '10, ACM, New York, NY, USA (2010), <http://doi.acm.org/10.1145/1882291.1882320>
15. Li, G., Li, P., Sawaya, G., Gopalakrishnan, G., Ghosh, I., Rajan, S.P.: GKLEE: Concolic verification and test generation for GPUs. In: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 215–224. PPOPP '12, ACM, New York, NY, USA (2012), <http://doi.acm.org/10.1145/2145816.2145844>
16. Microsoft: C++ Accelerated Massive Parallelism, <https://msdn.microsoft.com/en-us/library/hh265137.aspx>
17. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer Publishing Company, Incorporated (2010)
18. Nvidia: CUDA C Programming Guide v7.5, <http://docs.nvidia.com/cuda/cuda-c-programming-guide/>
19. OpenACC-standard.org: OpenACC: Directives for Accelerators, <http://www.openacc.org/>
20. Sung, I.J., Stratton, J.A., Hwu, W.M.W.: Data layout transformation exploiting memory-level parallelism in structured grid many-core applications. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques. pp. 513–522. PACT '10, ACM, New York, NY, USA (2010), <http://doi.acm.org/10.1145/1854273.1854336>
21. Ueng, S.Z., Lathara, M., Bagsorkhi, S.S., Hwu, W.m.W.: CUDA-Lite: Reducing GPU Programming Complexity, pp. 1–15. Springer Berlin Heidelberg, Berlin, Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-89740-8_1
22. Verdoolaege, S., Carlos Juega, J., Cohen, A., Ignacio Gómez, J., Tenllado, C., Catthoor, F.: Polyhedral parallel code generation for CUDA. ACM Trans. Archit. Code Optim. 9(4), 54:1–54:23 (Jan 2013), <http://doi.acm.org/10.1145/2400682.2400713>
23. Wu, J., Belevich, A., Bendersky, E., Heffernan, M., Leary, C., Pienaar, J., Roune, B., Springer, R., Weng, X., Hundt, R.: Gpucc: An open-source GPGPU compiler. In: Proceedings of the 2016 International Symposium on Code Generation and Optimization. pp. 105–116. CGO '16, ACM, New York, NY, USA (2016), <http://doi.acm.org/10.1145/2854038.2854041>
24. Yang, Y., Xiang, P., Kong, J., Zhou, H.: A GPGPU compiler for memory optimization and parallelism management. In: Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 86–97. PLDI '10, ACM, New York, NY, USA (2010), <http://doi.acm.org/10.1145/1806596.1806606>