Block-Size Independence for GPU Programs

Rajeev Alur, Joseph Devietti and Nimit Singhania

University of Pennsylvania

Abstract. Optimizing GPU programs by tuning execution parameters is essential to realizing the full performance potential of GPU hardware. However, many of these optimizations do not ensure correctness and subtle errors can enter while optimizing a GPU program. Further, lack of formal models and the presence of non-trivial transformations prevent verification of optimizations.

In this work, we verify transformations involved in tuning the execution parameter, *block-size*. First, we present a formal programming and execution model for GPUs, and then formalize *block-size independence* of GPU programs, which ensures tuning block-size preserves program semantics. Next, we present an inter-procedural analysis to verify block-size independence for synchronization-free GPU programs. Finally, we evaluate the analysis on the Nvidia CUDA SDK samples, where 35 global kernels are verified to be block-size independent.

1 Introduction

Graphics Processing Units (GPUs) have emerged as an important data-parallel compute platform. They are high-throughput, scalable, and useful for a wide variety of data-intensive applications like deep learning, virtual reality, and bioinformatics. However, programmers often struggle with tuning their GPU applications. The programmer has to repeatedly tune various execution parameters and rewrite parts of the program to achieve significant speedups compared to the CPU version. To add to the programmer's burden, performance is often not portable and the application needs to be re-tuned for another GPU.

Tuning GPU applications can introduce subtle errors into the application which can be difficult to debug and resolve. We need tools that can automatically detect such errors and ensure transformations performed while tuning an application are correct. Existing tools for GPU verification help identify correctness issues like data-races and barrier-divergence [12, 11, 2], but none verify correctness of transformations. Furthermore, synthesizing optimal execution configuration at compile-time is difficult since the optimization space is large and non-convex [21]. Hence, tuning applications by trying out different values for parameters is unavoidable. This makes it essential to have automatic tools to verify the correctness of transformations.

In this work, we focus on the correctness of tuning an execution parameter, *block-size*. A GPU program consists of a large number of threads that execute the same sequence of instructions. The threads are organized in a two-level hierarchy

where individual threads are grouped into thread-blocks and the thread-blocks together form a thread-grid. The parameter *block-size* represents the number of threads in each thread-block and is specified during program invocation, along with the total number of threads. The block-size determines how resources required by the program are allocated on GPU cores, and is often tuned to maximally utilize each core for performance while balancing performance across cores in a GPU. For instance, a 75% improvement in performance is achieved for a benchmark "SobolQRNG" on tuning block-size from 64 to 256.

We present an analysis to verify *block-size independence* of GPU programs which ensures modifying block-size is a valid transformation and does not introduce errors into the program. In the GPU execution model, sharing of data is permitted between threads of a thread-block, and changing the block-size alters the sets of threads allowed to share data, making program equivalence hard to reason about. Therefore, we only consider *synchronization-free* programs, where each thread executes independently of the other threads, and any sharing of data between threads is prohibited and leads to a data-race.

For synchronization-free programs, the analysis only needs to ensure that the execution of each thread is independent of block-size. Each thread in a GPU program is provided with a block-id, bid, a thread-id within the block, tid, and the block-size, bdim, which helps distinguish its execution from other threads. These values get modified when the block-size is modified, and the analysis tracks the flow of these values through variables in the program. Interestingly, the expression (bid.bdim + tid) identifies a globally unique id for each thread, and remains unchanged when the block-size is modified. Hence, the analysis further tracks the sub-expressions of this expression and whenever a variable is observed to be a function of this expression, it is marked independent of block-size. Further, to gain precision, the analysis also tracks block-size independent *multipliers*, so that expressions of the form (k.bid.bdim+k.tid), where k is a block-size independent value, can be proven block-size independent. Finally, if none of the block-size dependent values flow into the final state of any thread, then the program is block-size independent. The analysis uses a novel abstraction to track these values, where the *symbolic* constants track multipliers while the *abstract* constants track sub-expressions. This combination of abstract interpretation [18, 7] with symbolic execution [8, 3] helps scale the analysis while retaining good precision.

To understand this further, consider the function `cudaProcess()` in Figure 1 from a GPU program 'simpleCUDA2GL'. The function initializes pixels in an image represented by the array *g_odata*. Each thread initializes a globally unique location (x, y) with a value that is only a function of these coordinates. The coordinates x and y are independent of block-size. Also, the function is synchronization-free and each thread executes independently. Therefore, the function must be block-size independent. To prove this, the analysis tracks the flow of block-size dependent values bid, bdim, and tid through program variables. Note that, to mirror the 2-dimensional nature of the image, the threads are organized in a 2-dimensional grid, where the first and second dimensions identify

```
__global__ void cudaProcess(unsigned *g_odata, int imgw) {
    int tx = tid[0]; int ty = tid[1];
    int bw = bdim[0]; int bh = bdim[1];
    int x = bid[0]*bw + tx;
    int y = bid[1]*bh + ty;
    uchar4 c4 = make_uchar4((x & 0x20)?100:0, 0,
                            (y & 0x20)?100:0, 0);
    g_odata[y*imgw+x] = rgbToInt(c4.z, c4.y, c4.x);
}
```

Fig. 1: Example illustrating Block-Size Independence.

the x and y coordinates, respectively. During the analysis run, *imgw* is first assigned a block-size independent value. Next, *tx* is assigned tid_0, *ty* is assigned tid_1 and so on. Importantly, variables x and y are assigned $(\mathsf{bid}_0.\mathsf{bdim}_0 + \mathsf{tid}_0)$ and $(\mathsf{bid}_1.\mathsf{bdim}_1 + \mathsf{tid}_1)$, respectively, both of which are block-size independent. Further, calls to functions `make_uchar4` and `rgbToInt` return block-size independent values. Therefore, writes to array *g_odata* by threads are block-size independent, and the analysis verifies the program to be block-size independent.

We have implemented our tool in the LLVM open-source compiler. We implement an inter-procedural analysis and evaluate it on 34 sample programs from the Nvidia CUDA SDK 8.0 [19] samples. We observe that a large number of programs are synchronization-free and can be proven block-size independent. A few programs were trivially fixed to be block-size independent. Overall, the analysis verifies a total of 35 global kernels in 11 programs to be block-size independent, where a *global kernel* is an independent unit of execution in a GPU application. To summarize, the paper makes the following contributions.

- Identifies and formalizes the problem of *block-size independence* for GPU programs (Section 2).
- Presents a *scalable* inter-procedural analysis to verify block-size independence for the class of *synchronization-free* GPU programs (Section 3).
- Demonstrates the relevance of the problem through an extensive evaluation on the Nvidia CUDA SDK 8.0 samples (Section 4).

Lastly, we present some related work in Section 5 and conclude in Section 6.

2 Formalization

In this section, we present a formalization for the problem of block-size independence. We first define a formal semantics for the GPU programming model (Section 2.1) and the GPU execution model (Section 2.2). This establishes a framework under which we can reason about the correctness of transformations. We formalize block-size independence in Section 2.3. Finally, we discuss some design choices and limitations for the above formalization (Section 2.4).

2.1 GPU Programming Model

GPUs follow a Single Instruction Multiple Threads (SIMT) programming model, where a large number of threads execute the same sequence of instructions, called kernels. The threads are organized in a two-level hierarchy, where a set of threads form a thread-block, and set of blocks forms a thread-grid. The thread-grid can be multi-dimensional, where each thread is assigned a multi-dimensional thread-id and block-id. Further, threads have access to thread-private *local* memory, block-level *shared* memory, and a grid-level *global* memory. Each thread has access to its thread-id (tid), block-id (bid), number of threads per block (bdim) and the total number of threads (gdim). Finally, threads within a block can synchronize via a `__syncthreads()` barrier.

Formally, a GPU program is the tuple $\langle d, V_L, V_S, V_G, C, K\rangle$, where d represents the number of dimensions in the thread-grid, V_L, V_S, V_G represent the sets of local, shared and global variables in the program, $C = \{\mathsf{tid}, \mathsf{bid}, \mathsf{bdim}, \mathsf{gdim}\}$ represents a set of local constants, and K is the *kernel* or the sequence of instructions executed by each thread. Let $l \in V_L$ be a local variable and $v \in V_S \cup V_G$ be a shared/global array. Let E be a computable expression. The kernel K is defined by the grammar:

$$S := AS \mid \textbf{if}\ \langle test\rangle\ \textbf{then}\ S_1\ \textbf{else}\ S_2 \mid \textbf{while}\ \langle test\rangle\ \textbf{do}\ S \mid \texttt{\_\_syncthreads()} \mid S_1; S_2$$

$$\begin{aligned} AS := {} & [l := E(l_0, \ldots, l_n)] && \text{local assignments} \\ \mid {} & [l := v[l_0, \ldots, l_n]] && \text{multi-dimensional array reads} \\ \mid {} & [v[l_0, \ldots, l_n] := l] && \text{multi-dimensional array writes} \end{aligned}$$

Thread-Grid. Given the total number of threads (*i.e.* grid-size) and the number of threads per block (*i.e.* block-size), represented by d-dimensional vectors $\boldsymbol{N}$ and $\boldsymbol{B}$ respectively, we define the structure for the thread-grid. The thread-grid is d-dimensional where each dimension i is divided into $\lceil \boldsymbol{N}_i/\boldsymbol{B}_i \rceil$ blocks. The total number of threads along ith dimension is $\boldsymbol{N}_i$, and therefore, the first $(\lceil \boldsymbol{N}_i/\boldsymbol{B}_i \rceil - 1)$ blocks consist of $\boldsymbol{B}_i$ threads, whereas the last block consists of $(\boldsymbol{N}_i - (\lceil \boldsymbol{N}_i/\boldsymbol{B}_i \rceil - 1)\boldsymbol{B}_i)$ threads. The blocks and threads are assigned a d-dimensional block-id $\boldsymbol{b}$ and thread-id $\boldsymbol{t}$. The block-ids range from 0 to $\lceil \boldsymbol{N}_i/\boldsymbol{B}_i \rceil - 1$ for each dimension i, while the thread-ids range from 0 to $\min(\boldsymbol{B}_i, \boldsymbol{N}_i - \boldsymbol{b}_i\boldsymbol{B}) - 1$ and identify the positions of threads within their blocks.

Figure 2 presents an example 2-dimensional thread-grid with 22×10 total threads, with 4×3 threads per block. There are in total 6×4 blocks. Also, the last block along each dimension has fewer threads than the first few blocks to preserve the total number of threads along the dimension.

2.2 GPU Execution Model

We next present the semantics of executing a GPU program. Given a global state σ^G that maps each global variable to a specific value, and a thread-grid configuration, given by the grid-size $\boldsymbol{N}$ and the block-size $\boldsymbol{B}$, let $[\![K]\!]^G(\sigma^G, \boldsymbol{N}, \boldsymbol{B})$

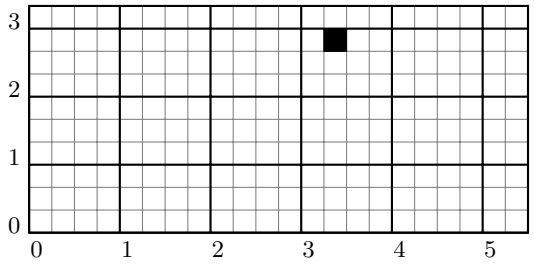


Fig. 2: An example 2-dimensional thread-grid with 22×10 total threads and 4×3 threads per block. Each solid block represents a thread-block, while each cell represents a thread. The darkened cell corresponds to a thread with block-id $(3, 2)$ and thread-id $(1, 2)$.

represent the global state obtained after the execution of kernel K. Let τ represent a thread in the thread-grid. Let $\boldsymbol{t} = \mathsf{tid}(\tau)$ and $\boldsymbol{b} = \mathsf{bid}(\tau)$ be the thread-id and block-id for the thread. Let $\mathcal{G}(\boldsymbol{N}, \boldsymbol{B})$ represent the set of all threads in the grid. Let $T(\boldsymbol{b}, \boldsymbol{N}, \boldsymbol{B})$ be the set of all threads with block-id $\boldsymbol{b}$, *i.e.* $\{\tau \in \mathcal{G}(\boldsymbol{N}, \boldsymbol{B}) : \mathsf{bid}(\tau) = \boldsymbol{b}\}$. We first present the semantics for executing threads within a block, and then the semantics of composing executions for blocks.

Fine-grained execution for threads. We use a fine-grained semantics to execute threads within a block, where all threads execute instructions in lock-step. Given a block-id $\boldsymbol{b}$, the threads in the block are $T = T(\boldsymbol{b}, \boldsymbol{N}, \boldsymbol{B})$. We simultaneously maintain state for all threads. Each thread has access to a private copy of local variables and a common copy of shared and global variables. Therefore, the execution state σ consists of the local state $\sigma^L : V_L \times T \to \mathcal{V}$ that maps local variables in each thread to their values, the shared state $\sigma^S : V_S \to \mathcal{V}$ that maps shared variables to their values, and the global state $\sigma^G : V_G \to \mathcal{V}$ that maps global variables to their values. Further for each thread $\tau \in T$, the local constants $\mathsf{bdim}(\tau)$ and $\mathsf{gdim}(\tau)$ are assigned block-size $\boldsymbol{B}$ and grid-size $\boldsymbol{N}$, respectively. We now present the semantics. Let $[\![S]\!](\sigma, \Pi)$ represent the execution of a statement S for a set of threads Π starting in state σ. The semantics of executing kernel K for threads with block-id $\boldsymbol{b}$ is given by $[\![K]\!](\sigma, T(\boldsymbol{b}, \boldsymbol{N}, \boldsymbol{B}))$. Note the resulting state consists of all variables and not just global variables. We define semantics by structural induction on S.

Assignments. We first define semantics of executing an assignment statement for a single thread τ. Let $\sigma' \equiv [\![AS]\!](\sigma, \tau)$ represent the semantics. For local computations, $[l := E(l_0, \ldots, l_n)]$, the semantics updates the value of l in state σ' to the value $E(\sigma(l_0, \tau), \ldots, \sigma(l_n, \tau))$. For array reads, $[l := v[l_0, \ldots, l_n]]$, the semantics updates the value of variable l with the value at location $\boldsymbol{x} = (\sigma(l_0, \tau), \sigma(l_1, \tau), \ldots, \sigma(l_n, \tau))$ in array v, *i.e.* $\sigma'(l, \tau) = \sigma(v)(\boldsymbol{x})$. Finally, for array writes, $[v[l_0, \ldots, l_n] := v]$, the semantics updates the value at location

$\boldsymbol{x} = (\sigma(l_0,\tau), \sigma(l_1,\tau), \ldots, \sigma(l_n,\tau))$ in array v to the value $\sigma(l,\tau)$:

$$\sigma'(v)(\boldsymbol{x}) = \sigma(l,\tau), \text{ and for all } \boldsymbol{y} \neq \boldsymbol{x}, \sigma'(v)(\boldsymbol{y}) = \sigma(v)(\boldsymbol{y}).$$

Note that the constants $\mathsf{tid}, \mathsf{bid}, \mathsf{bdim}$, and gdim can also appear on the right-hand side of these assignments. Next, we present the semantics of executing the assignment for a set of threads $\Pi = \{\tau_0, \ldots, \tau_n\}$. The semantics sequentially compose the execution of individual threads, ordered by their thread-ids. Hence, $[\![AS]\!](\sigma, \{\}) = \sigma$, and for all $0 \leq i \leq n$,

$$[\![AS]\!](\sigma, \{\tau_i, \ldots, \tau_n\}) = [\![AS]\!]([\![AS]\!](\sigma, \tau_i), \{\tau_{i+1}, \ldots, \tau_n\}).$$

Sequences. The semantics of executing sequence of statements $S_1; S_2$ consists of executing S_1 for all threads, followed by executing S_2:

$$[\![S_1; S_2]\!](\sigma, \Pi) = [\![S_2]\!]([\![S_1]\!](\sigma, \Pi), \Pi).$$

Conditionals. The semantics for conditionals *serializes* the execution of the two branches. First all threads for which the test, given by a local boolean variable l, is true, execute S_1. Then the remaining threads execute S_2 to produce the desired state. Let $\Pi_1 = \{\tau \in \Pi : \sigma(l,\tau) = \mathsf{true}\}$. The semantics are:

$$[\![\textbf{if } l \textbf{ then } S_1 \textbf{ else } S_2]\!](\sigma, \Pi) = [\![S_2]\!]([\![S_1]\!](\sigma, \Pi_1), \Pi \setminus \Pi_1).$$

Loops. The semantics for loops are similar to that for conditionals, except the execution repeats until the test condition, given by a local boolean variable, becomes false for all threads. Let $\sigma_0, \ldots, \sigma_n$ and $\Pi_0, \ldots, \Pi_n$ be a series of states and sets of threads, such that $\sigma_0 = \sigma$, $\Pi_0 = \{\tau \in \Pi : \sigma(l,\tau) = \mathsf{true}\}$, $\sigma_i = [\![S]\!](\sigma_{i-1}, \Pi_{i-1})$, $\Pi_i = \{\tau \in \Pi_{i-1} : \sigma_i(l,\tau) = \mathsf{true}\}$, and $\Pi_n = \{\}$. If such a series exists, then the final state σ_n is the desired result of executing the loop.

Syncthreads. Due to the lock-step execution of threads, the `__syncthreads()` barrier does not need special semantics and returns the initial state σ.

Coarse-grained execution for blocks. We next present the semantics of composing executions of individual blocks. We present a coarse-grained semantics where each block executes independently and the final state is obtained by sequentially composing executions of individual blocks, ordered by their block-ids. The blocks share only the global variables, and the local and shared variables are initialized to undefined values before the execution for a block begins and discarded after the execution ends. Let $[\![K]\!]^G(\sigma^G, \Gamma, \boldsymbol{N}, \boldsymbol{B})$ represent the execution for blocks with block-ids in $\Gamma = \{\boldsymbol{b}_0, \ldots, \boldsymbol{b}_n\}$ starting in initial global state σ^G. We define it as follows. First, $[\![K]\!]^G(\sigma^G, \{\}, \boldsymbol{N}, \boldsymbol{B}) = \sigma^G$. Next, for all $0 \leq i \leq n$,

$$[\![K]\!]^G(\sigma^G, \{\boldsymbol{b}_i, \ldots, \boldsymbol{b}_n\}, \boldsymbol{N}, \boldsymbol{B}) = [\![K]\!]^G(\mathsf{Proj}(\sigma', V_G), \{\boldsymbol{b}_{i+1}, \ldots, \boldsymbol{b}_n\}, \boldsymbol{N}, \boldsymbol{B}),$$
$$\text{where } \sigma' = [\![K]\!](\sigma^G \cup \sigma^S_\perp \cup \sigma^L_\perp, T(\boldsymbol{b}_i, \boldsymbol{N}, \boldsymbol{B})),$$

and $\sigma^L_\perp$ and $\sigma^S_\perp$ are local and shared states with undefined values, while $\mathsf{Proj}(\sigma, V)$ projects the state σ onto the variables in set V. Note that the final state consists only of the global variables. Now the desired state after the execution of the GPU program, $[\![K]\!]^G(\sigma^G, \boldsymbol{N}, \boldsymbol{B})$, is given by $[\![K]\!]^G(\sigma^G, \mathcal{B}, \boldsymbol{N}, \boldsymbol{B})$ where $\mathcal{B}$ is the set of all block-ids in the thread-grid.

Invalidation Semantics. We next describe scenarios under which the execution of the program is erroneous and produces an error state $\perp$. First, a *data-race* between threads leads to an error state. A data-race occurs when two threads access the same shared/global memory location, and the execution of the accesses is not separated by a `__syncthreads()` barrier. Second, an execution where only few of the threads within a block reach a `__syncthreads()` barrier produces an error state, and is called a *barrier divergence*. These semantics help incorporate features of the general GPU execution model into the formalization.

To keep the execution model simple, we discuss the invalidation semantics informally. The focus of the paper is on proving *functional* equivalence of the original and the transformed program. For such a property, precise invalidation semantics are not necessary. We still rely on the data-race freedom of programs to prove the correctness of our analysis. However, the informal nature of the semantics suffices.

2.3 Block-Size Independence

We now define the block-size independence for a GPU program. Let two states σ and σ' be equivalent ($\sigma \equiv \sigma'$), if they consist of the same set of variables and each variable has the same valuation in both states. We state the formal definition here.

Definition 1. *A GPU program* $\langle d, V_L, V_S, V_G, C, K\rangle$ *is* block-size independent*, iff for all initial global states* σ^G *and grid-sizes* $\boldsymbol{N}$*, the execution of the program is independent of the block-size* $\boldsymbol{B}$*, that is:*

$$\textit{for all } \sigma^G, \boldsymbol{N}, \boldsymbol{B}, \boldsymbol{B}',\ [\![K]\!]^G(\sigma^G, \boldsymbol{N}, \boldsymbol{B}) \equiv [\![K]\!]^G(\sigma^G, \boldsymbol{N}, \boldsymbol{B}').$$

2.4 Discussion

We have presented so far a formal programming and execution model for GPU programs and defined block-size independence with respect to this model. The proposed model closely follows popular programming models like CUDA and OpenCL. However, there are few restrictions and limitations in the proposed model that we discuss here:

Lock-step execution. We use a simplified execution model where we assume all threads in a block to execute in lock-step. This is not true in practice for performance reasons. However, we are only concerned with the functional behavior of programs and proving functional correctness of block-size transformation. Also, the simplified execution model is functionally equivalent to the model used in practice when programs are free of data-races.

Data-race freedom. Our formalization assumes that the GPU program being transformed is free of data-races and other such correctness issues. These issues have been tackled previously [2, 11, 12], and therefore, we focus only on the correctness of block-size transformation.

Total number of threads. In our formalization, we specify the number of threads $\boldsymbol{N}$ as one of the invocation parameters. Among the popular models, OpenCL [24] closely follows this model. CUDA [17], however, specifies the number of blocks $\boldsymbol{N_b}$ as an invocation parameter and computes the number of threads along ith dimension as $\boldsymbol{B}_i.(\boldsymbol{N_b})_i$ *i.e.* the product between the number of blocks and the block-size. However, specifying the number of threads $\boldsymbol{N}$ provides more flexibility in defining the total number of threads. Also, the total number of threads remains unchanged when the block-size is modified, which makes proving program equivalence easier. Further, when the new block-size $\boldsymbol{B'}$ is a divisor the number of threads $\boldsymbol{N}$ along each grid-dimension, our model is also applicable to CUDA and the new number of blocks along ith dimension can be computed as $\boldsymbol{B}_i.(\boldsymbol{N_b})_i/\boldsymbol{B'}_i$.

Structures and pointers. Our formal model only considers scalars and arrays, while the general models CUDA and OpenCL also support structures and pointers. The key insights for arrays carry over to structures and pointers, and therefore for simplicity, we omit them from our model. We address these, however, in the implementation of our analysis.

3 Analysis for Synchronization-free GPU Programs

This section presents an analysis to verify block-size independence for *synchronization-free* GPU programs, where the kernel does not consist of `__syncthreads()` barriers. In a synchronization-free GPU program, each thread must execute independently of the other threads (since any dependence on updates from other threads leads to a data-race). Therefore, the *global* problem of verifying block-size independence of the program reduces to the *local* problem of verifying block-size independence for the execution of each thread in the program (Section 3.1).

Next, the execution of a thread is independent of block-size if the writes by the thread to the shared and global variables do not depend on block-size.[1] A write can depend on block-size if either the location accessed, the value written or the condition under which the write is executed is dependent on block-size. The only sources of block-size dependence in a thread are the thread's block-id, $\mathsf{bid}(\tau)$, the thread-id, $\mathsf{tid}(\tau)$, and the block-size itself, $\mathsf{bdim}(\tau) = \boldsymbol{B}$. Further, the expression $\mathsf{gid}(\tau) = (\mathsf{bid}.\mathsf{bdim} + \mathsf{tid})(\tau)$ is independent of block-size. This is because $\mathsf{gid}(\tau)$ identifies a unique global location of the thread in the thread-grid and remains unchanged when the block-size is modified. For example in Figure 2, the thread with thread-id $(1, 2)$ and block-id $(3, 2)$ has a unique global-id $(3.4 + 1, 2.3 + 2) = (13, 8)$, which remains unchanged for all block-sizes. We incorporate these features into our analysis to check block-size independence for each thread (Section 3.2).

[1] Reads can be ignored because our `__syncthreads()`-free and race-free assumptions permit a thread to only read values it has written itself or are part of the initial state.

3.1 Reduction to Thread-local Block-Size Independence

We first define *thread-local block-size independence* for GPU programs. A GPU program is thread-local block-size independent if the execution of each thread in the thread-grid is independent of block-size. Given block-sizes $\boldsymbol{B}$ and $\boldsymbol{B}'$, let a thread τ in grid $\mathcal{G}(\boldsymbol{N}, \boldsymbol{B})$ be equivalent to another thread τ' in grid $\mathcal{G}(\boldsymbol{N}, \boldsymbol{B}')$, *i.e.* $\tau \equiv \tau'$, if they have the same unique global location in the thread-grid, namely:

$$\text{for all } 0 \leq i < d,\ (\mathsf{bid}_i(\tau).\boldsymbol{B}_i + \mathsf{tid}_i(\tau)) = (\mathsf{bid}_i(\tau').\boldsymbol{B}'_i + \mathsf{tid}_i(\tau'))$$

We observe this to be a one-to-one relation, where each thread τ in the first grid corresponds to a *unique* global thread τ' in the second grid. Now, the program is thread-local block-size independent, if each pair of equivalent global threads has equivalent executions. Recall $[\![S]\!](\sigma, \Pi)$ denotes the execution of statement S for a set of threads Π starting in initial state σ.

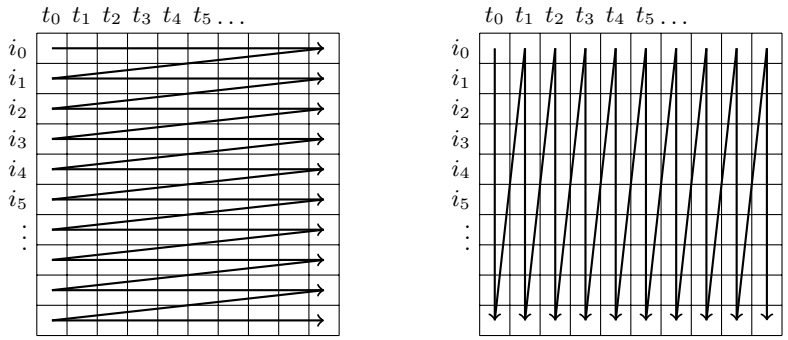
Definition 2. *A GPU program* $\langle d, V_L, V_S, V_G, C, K\rangle$ *is* thread-local block-size independent*, iff for all initial states* σ^G *and grid-sizes* $\boldsymbol{N}$*, the global state after the execution of a thread in the thread-grid is independent of block-size, where the local and shared variables are initialized to undefined values. Formally, the program is thread-local block-size independent iff:*

$$\begin{aligned}&\textit{for all } \sigma^G, \boldsymbol{N}, \boldsymbol{B}, \boldsymbol{B}', \tau \in \mathcal{G}(\boldsymbol{N}, \boldsymbol{B}), \tau' \in \mathcal{G}(\boldsymbol{N}, \boldsymbol{B}'),\\ &\qquad \tau \equiv \tau' \implies \mathsf{Proj}([\![K]\!](\sigma, \{\tau\}), V_G) \equiv \mathsf{Proj}([\![K]\!](\sigma, \{\tau'\}), V_G),\\ &\qquad\qquad \textit{where } \sigma \equiv (\sigma^L_\perp \cup \sigma^S_\perp \cup \sigma^G).\end{aligned}$$

We show that if a GPU program is synchronization-free, verifying thread-local block-size independence is sufficient to verify block-size independence for the program. We first observe that for a synchronization-free program, the lock-step execution of threads in a block is equivalent to executing threads one after another. This is because, to avoid data-races, each thread must operate independently and not see updates from other threads. Therefore, the order of execution between threads does not matter and a fine-grained interleaving (Figure 3a) produces the same execution as a coarse-grained interleaving (Figure 3b).

Lemma 1. *Given a synchronization-free GPU program* $\langle d, V_L, V_S, V_G, C, K\rangle$ *and a set of threads* $\Pi = \{\tau_0, \ldots, \tau_k\}$*, the lock-step execution of threads is equivalent to executing threads sequentially:*

$$\begin{aligned}&\textit{for all } \sigma, \Pi,\ [\![K]\!](\sigma, \Pi) \equiv \sigma_{k+1},\\ &\qquad\qquad \textit{where } \sigma_0 = \sigma \textit{ and } \sigma_{i+1} = [\![K]\!](\sigma_i, \{\tau_i\}) \textit{ for all } 0 \leq i \leq k.\end{aligned}$$



(a) Fine-grained interleaving (b) Coarse-grained interleaving

Fig. 3: The figure shows fine-grained vs coarse-grained interleaving of threads in a block. The rows represent sequence of instructions to be executed, while the columns represent the threads in a block. The arrows signify the order in which the threads and the instructions are executed.

By Lemma 1, the lock-step execution of threads in a block can be substituted with sequential execution of threads. Next, we observe that we can execute each thread in a state where the local and shared variables are undefined initially. This is because, the thread must not observe any updates to these variables from the previously executed threads, or we would have a data-race. Also, these variables are discarded at the end of the execution of the block, and we need not retain their values. Remember $[\![K]\!]^G(\sigma^G, \Gamma, \boldsymbol{N}, \boldsymbol{B})$ represents execution of a set of blocks Γ, where the shared and local variables are undefined initially and the result of the execution consists only of the global state.

Lemma 2. *Given a synchronization-free GPU program $\langle d, V_L, V_S, V_G, C, K\rangle$ and a block-id $\boldsymbol{b}$, the lock-step execution for block $\boldsymbol{b}$ is equivalent to executing threads sequentially, with local and shared variables initialized to undefined values:*

$$\begin{aligned}&\textit{for all } \sigma^G, \boldsymbol{b}, \boldsymbol{N}, \boldsymbol{B},\ [\![K]\!]^G(\sigma^G, \{\boldsymbol{b}\}, \boldsymbol{N}, \boldsymbol{B}) \equiv \sigma^G_{k+1},\\ &\qquad \textit{where } \sigma^G_0 = \sigma^G \textit{ and } \sigma^G_{i+1} = \mathsf{Proj}([\![K]\!](\sigma^L_\perp \cup \sigma^S_\perp \cup \sigma^G_i, \{\tau_i\}), V_G),\\ &\qquad\qquad \textit{for all } \tau_i \textit{ in } T(\boldsymbol{b}, \boldsymbol{N}, \boldsymbol{B}).\end{aligned}$$

Finally from Lemma 2, the execution of each thread in the first grid can be substituted with the execution of equivalent thread in the second grid, and therefore, thread-local block-size independence of a synchronization-free program implies block-size independence for the program. We conclude the following theorem.

Theorem 1. *If a synchronization-free GPU program $\langle d, V_L, V_S, V_G, C, K\rangle$ is* thread-local block-size independent*, then it is also* block-size independent*.*

3.2 Analysis

We present our analysis to check thread-local block-size independence of GPU programs and to ensure that the execution of each thread is block-size independent. Initially when a thread's execution starts, only constants $\mathsf{bid}, \mathsf{bdim}$ and tid are block-size dependent and the remaining variables are block-size independent. While bdim is equal to block-size, the thread-id tid and block-id bid of a thread also depend on the block-size and get updated when the block-size is modified. Hence, if any of these values potentially flows into a global variable update, then the final global state after the thread's execution depends on block-size and the program is block-size dependent. The analysis defines an abstraction of state and abstract semantics for kernel instructions to track the flow of block-size dependent values during a thread's execution. Note that we run the analysis and show the block-size independence separately for each dimension of thread-grid. So for the subsequent discussion, consider bid, bdim and tid to be one-dimensional values. This is not too restrictive, since most programs are block-size independent with respect to each grid-dimension. Also, this greatly simplifies the analysis both in its complexity and running time.

Abstraction. The analysis defines an abstraction of program state to track dependence of local scalar variables on block-size. Let $\hat{\sigma}$ be the abstraction of the program state σ, which maps each local variable to an abstract value, *i.e.* $V_L \to \hat{\mathcal{V}}$. Let l, k_0 be local variables. Let f_0 be a function that maps each thread to a block-size independent value. For integer and real variables, the abstraction is defined as:

$$\hat{\sigma}(l) = \begin{cases} \mathsf{c_{ind}}, & \text{for all } \tau,\ \sigma(l,\tau) = f_0(\tau). \\ k_0\mathsf{c_{bid}}, & \hat{\sigma}(k_0) = \mathsf{c_{ind}}; \text{for all } \tau,\ \sigma(l,\tau) = \sigma(k_0,\tau).\mathsf{bid}(\tau). \\ k_0\mathsf{c_{bdim}}, & \hat{\sigma}(k_0) = \mathsf{c_{ind}}; \text{for all } \tau,\ \sigma(l,\tau) = \sigma(k_0,\tau).\mathsf{bdim}(\tau). \\ k_0\mathsf{c_{tid}}, & \hat{\sigma}(k_0) = \mathsf{c_{ind}}; \text{for all } \tau,\ \sigma(l,\tau) = \sigma(k_0,\tau).\mathsf{tid}(\tau) + f_0(\tau). \\ k_0\mathsf{c_{bid}}\mathsf{c_{bdim}}, & \hat{\sigma}(k_0) = \mathsf{c_{ind}}; \\ & \text{for all } \tau,\ \sigma(l,\tau) = \sigma(k_0,\tau).\mathsf{bid}(\tau).\mathsf{bdim}(\tau) + f_0(\tau). \\ \mathsf{c_{bsize}}, & \text{otherwise.} \end{cases}$$

The value $\mathsf{c_{ind}}$ represents all block-size independent values. The abstract value $\mathsf{c_{bsize}}$ represents values with arbitrary dependence on block-size. We observe the expression $(k_0.\mathsf{bid}.\mathsf{bdim} + k_0.\mathsf{tid})$, where k_0 is a block-size independent variable, is independent of block-size. To take this account, our abstraction tracks different sub-expressions of this expression, $k_0\mathsf{c_{bid}}, k_0\mathsf{c_{bdim}}, k_0\mathsf{c_{tid}}$, and $k_0\mathsf{c_{bid}}\mathsf{c_{bdim}}$, where k_0 is the *multiplier* or a symbolic constant representing a block-size independent local variable. We assume each local variable has a unique definition (*e.g.* SSA form), and the variables are not updated after they are first defined. Hence, the symbolic constant truly represents the variable used as multiplier in the abstract value, and we do not differentiate between the variable and the symbolic constant representing the variable.

We similarly define an abstraction for local boolean variables, which tracks dependence of the condition on block-size. Let b_0 be a block-size independent

$$\textsc{Sum1}\ \frac{l := l_0 + l_1 \quad \hat{\sigma}(l_0) = \mathsf{c_{ind}} \quad \hat{\sigma}(l_1) \in \{k_0\mathsf{c_{tid}}, k_1\mathsf{c_{bid}}\mathsf{c_{bdim}}\}}{\hat{\sigma}'(l) := \hat{\sigma}(l_1)} \qquad \textsc{Sum2}\ \frac{l := l_0 + l_1 \quad \hat{\sigma}(l_0) = k_0\mathsf{c_{tid}} \quad \hat{\sigma}(l_1) = k_1\mathsf{c_{bid}}\mathsf{c_{bdim}} \quad k_0 \equiv k_1}{\hat{\sigma}'(l) := \mathsf{c_{ind}}}$$

$$\textsc{Prod1}\ \frac{l := l_0.l_1 \quad \hat{\sigma}(l_0) = \mathsf{c_{ind}} \quad \hat{\sigma}(l_1) \in \{\mathsf{c_{bid}}, \mathsf{c_{bdim}}, \mathsf{c_{tid}}, \mathsf{c_{bid}}\mathsf{c_{bdim}}\}}{\hat{\sigma}'(l) := l_0\hat{\sigma}(l_1)} \qquad \textsc{Prod2}\ \frac{l := l_0.l_1 \quad \hat{\sigma}(l_0) = \mathsf{c_{bid}} \quad \hat{\sigma}(l_1) = k_0\mathsf{c_{bdim}}}{\hat{\sigma}'(l) := k_0\mathsf{c_{bid}}\mathsf{c_{bdim}}}$$

$$\textsc{Prod3}\ \frac{l := l_0.l_1 \quad \hat{\sigma}(l_0) = k_0\mathsf{c_{bid}} \quad \hat{\sigma}(l_1) = \mathsf{c_{bdim}}}{\hat{\sigma}'(l) := k_0\mathsf{c_{bid}}\mathsf{c_{bdim}}} \qquad \textsc{Read}\ \frac{l := v[l_0, \ldots, l_n] \quad \hat{\sigma}(l_0) = \mathsf{c_{ind}} \ldots \hat{\sigma}(l_n) = \mathsf{c_{ind}}}{\hat{\sigma}'(l) := \mathsf{c_{ind}}}$$

$$\textsc{Arith}\ \frac{l := l_0 \text{ op } l_1 \quad \hat{\sigma}(l_0) = \mathsf{c_{ind}} \quad \hat{\sigma}(l_1) = \mathsf{c_{ind}}}{\hat{\sigma}'(l) := \mathsf{c_{ind}}} \qquad \textsc{Rel}\ \frac{l := l_0 \text{ rel } l_1 \quad \hat{\sigma}(l_0) = \mathsf{c_{ind}} \quad \hat{\sigma}(l_1) = \mathsf{c_{ind}}}{\hat{\sigma}'(l) := \mathsf{b_{ind}}} \qquad \textsc{Bool}\ \frac{l := l_0 \text{ bop } l_1 \quad \hat{\sigma}(l_0) = \mathsf{b_{ind}} \quad \hat{\sigma}(l_1) = \mathsf{b_{ind}}}{\hat{\sigma}'(l) := \mathsf{b_{ind}}}$$

Fig. 4: Abstract semantics for different assignment statements and initial abstract states. State $\hat{\sigma}$ is the incoming abstract state while $\hat{\sigma}'$ is the updated state after the assignment. The the rules are valid only when the path-predicate $\hat{\pi}$ is $\mathsf{b_{ind}}$. Lastly, *op*, *rel* and *bop* are arithmetic, relational and boolean operators, respectively.

boolean function. The abstraction for boolean variables is:

$$\hat{\sigma}(l) = \begin{cases} \mathsf{b_{ind}}, & \text{for all } \tau,\ \sigma(l,\tau) = b_0(\tau). \\ \mathsf{b_{bsize}}, & \text{otherwise.} \end{cases}$$

Finally, we do not track shared and global variables or arrays in our abstraction. We compensate by tracking each write to these variables and ensuring that the writes are independent of block-size.

We further define a *path-predicate*, $\hat{\pi}$, which is the condition under which a statement is executed. The value of $\hat{\pi}$ is an abstract boolean value, representing whether the condition is dependent on block-size or not.

Abstract Semantics. We now define some abstract semantics for propagating abstract state $\hat{\sigma}$ and path-predicate $\hat{\pi}$ through statements in the kernel. Figure 4 defines updates to abstract states for different assignment statements and initial states. Note the rules in Figure 4 are only valid if the path-predicate $\hat{\pi}$ is $\mathsf{b_{ind}}$. Also, we only show rules for scenarios where the result is non-trivial and not $\mathsf{c_{bsize}}/\mathsf{b_{bsize}}$. Otherwise, if $\hat{\pi} = \mathsf{b_{bsize}}$ or the rule is not shown, the updated value for arithmetic/boolean variables is $\mathsf{c_{bsize}}/\mathsf{b_{bsize}}$. The path-predicate remains unchanged after each statement, unless specified.

We now briefly describe the rules shown in Figure 4. Note when the multiplier k for an abstract value is constant 1, we drop the multiplier, *e.g.* $\mathsf{c_{bid}}$ in rule Prod1. The rules ensure that the abstraction is preserved. For example, in rule Sum2, abstract values $k_0\mathsf{c_{tid}}$ and $k_1\mathsf{c_{bid}}\mathsf{c_{bdim}}$ are added together, where k_0 equals

k_1. This is equivalent to the expression $(k_0.\mathsf{tid} + k_0.\mathsf{bid}.\mathsf{bdim})$, which we know is block-size independent. Hence, the final result is assigned the value $\mathsf{c}_{\mathsf{ind}}$. Similarly, the other rules update the abstract state while preserving the abstraction. An important point to note here is that during the product operation (rules Prod1, Prod2, Prod3), the multiplier for at least one of the operands must be constant 1, so that the multiplier for the other operand is set as the final multiplier. Otherwise, the result is set to $\mathsf{c}_{\mathsf{bsize}}$. This ensures that the set of symbolic values for the multiplier is limited to the set of variables in the program and we do not consider complex expressions on variables for the multiplier. While this is imprecise, it is necessary to scale the analysis.

We next consider writes to shared/global arrays $[v[l_0, \ldots, l_n] := l]$, where the analysis checks if the accessed location, the value written and the path-predicate are independent of block-size, *i.e.* the values $\hat{\sigma}(l_0), \ldots, \hat{\sigma}(l_n)$ and $\hat{\sigma}(l)$ must be $\mathsf{c}_{\mathsf{ind}}$ and the path-predicate $\hat{\pi}$ must be $\mathsf{b}_{\mathsf{ind}}$. If this is not the case, the write is potentially a function of block-size and the analysis reports the write, and the kernel itself, to be *block-size dependent*. This also ensures the values in shared/global arrays are always block-size independent, and thus, the array reads return a consistent value in rule Read in Figure 4.

For conditionals [**if** l **then** S_1 **else** S_2], the analysis sets the path-predicates for S_1 and S_2 to $(\hat{\pi} \wedge \hat{\sigma}(l))$ and $(\hat{\pi} \wedge \neg\hat{\sigma}(l))$, respectively, and propagates the same initial abstract state $\hat{\sigma}$ to both statements. Further, the final state after the conditional is a *merge* of states after S_1 and S_2. If the values for a variable are identical in both states (*i.e.* the type and the multiplier are equal), then this is set as the merged value for the variable. Otherwise, the merged value is set to $\mathsf{c}_{\mathsf{bsize}}/\mathsf{b}_{\mathsf{bsize}}$. The path-predicate after the conditional is the same as the predicate $\hat{\pi}$ before the conditional.

The semantics for loops are defined similarly to conditionals, but we must additionally ensure that the analysis terminates. We observe that the set of abstract values and the merge operation define a finite upper semi-lattice, with a small number of different value types and the multiplier ranging over the finite set of local variables. Further, the abstract semantics are monotonic over the semi-lattice. Therefore, the fixed point computation on loops must terminate.

Algorithm. The overall algorithm is as follows. We initialize local variables to $\mathsf{c}_{\mathsf{ind}}/\mathsf{b}_{\mathsf{ind}}$ in the initial abstract state $\hat{\sigma}$, while the path-predicate $\hat{\pi}$ is initialized to $\mathsf{b}_{\mathsf{ind}}$. The constants bid, bdim and tid are assigned values $\mathsf{c}_{\mathsf{bid}}$, $\mathsf{c}_{\mathsf{bdim}}$ and $\mathsf{c}_{\mathsf{tid}}$, respectively, while gdim is independent of block-size and assigned $\mathsf{c}_{\mathsf{ind}}$. The analysis executes the kernel for the abstract state $\hat{\sigma}$ and the path-predicate $\hat{\pi}$ with the abstract semantics defined above. If it encounters a potentially block-size dependent shared or global write, it terminates with block-size dependence. Otherwise, it reports the kernel to be block-size independent.

Inter-procedural analysis. Our analysis also supports inter-procedural analysis, where a kernel can call other kernels. We do a bottom-up traversal on the call-graph, where the callees are analyzed before the callers. We analyze each kernel assuming the parameters are set to $\mathsf{c}_{\mathsf{ind}}/\mathsf{b}_{\mathsf{ind}}$ initially and reuse this analysis result for all calls to the kernel with call arguments as $\mathsf{c}_{\mathsf{ind}}/\mathsf{b}_{\mathsf{ind}}$. For calls with

block-size dependent arguments, we conservatively report the call to be block-size dependent and return $\mathsf{c}_{\mathsf{bsize}}/\mathsf{b}_{\mathsf{bsize}}$. For library calls (where the source code is not linked) and inline assembly instructions, we conservatively assume the function to be block-size dependent and to return value $\mathsf{c}_{\mathsf{bsize}}/\mathsf{b}_{\mathsf{bsize}}$. However, for specific cases, like library calls to Math functions `__sinf`, `__cosf`, `__sqrtf` etc., where the result is a trivial function of inputs, we assume the call to be block-size independent, and also return $\mathsf{c}_{\mathsf{ind}}/\mathsf{b}_{\mathsf{ind}}$ if the call-arguments are $\mathsf{c}_{\mathsf{ind}}/\mathsf{b}_{\mathsf{ind}}$. Note that we do not support recursive procedures in our analysis, which are rarely present in GPU programs.

Example. We illustrate our analysis using the example in Figure 1. We run the analysis separately for the two thread-grid dimensions. For the first thread-grid dimension, the analysis initializes variables as $\hat{\sigma}(\mathsf{bid}_0) = \mathsf{c}_{\mathsf{bid}}, \hat{\sigma}(\mathsf{bdim}_0) = \mathsf{c}_{\mathsf{bdim}}, \hat{\sigma}(\mathsf{tid}_0) = \mathsf{c}_{\mathsf{tid}}, \hat{\sigma}(\mathsf{bid}_1) = \hat{\sigma}(\mathsf{bdim}_1) = \hat{\sigma}(\mathsf{tid}_1) = \hat{\sigma}(imgw) = \mathsf{c}_{\mathsf{ind}}$. Also, it initializes the path-condition to $\mathsf{b}_{\mathsf{ind}}$, which is never modified. Next, it executes the statement $[tx := \mathsf{tid}_0]$, and sets $\hat{\sigma}(tx)$ to $\mathsf{c}_{\mathsf{tid}}$. It similarly assigns values to variables ty, bw, bh. When computing x, it first computes the product $\mathsf{bid}_0.bw$ which is equal to $\mathsf{c}_{\mathsf{bid}}\mathsf{c}_{\mathsf{bdim}}$, and then computes x as the sum of values $\mathsf{c}_{\mathsf{bid}}\mathsf{c}_{\mathsf{bdim}}$ and $\mathsf{c}_{\mathsf{tid}}$, which we know is $\mathsf{c}_{\mathsf{ind}}$. The execution for the remaining statements continues similarly. Finally, the global write to image *g_odata* is executed with block-size independent abstract values and path-condition, and hence, the write is block-size independent. Therefore, the analysis declares the program block-size independent along this thread-grid dimension. The analysis repeats a similar process for the other thread-grid dimension and concludes the program to be block-size independent.

Implementation. We have implemented the analysis as a pass in LLVM compiler. We define the abstract domain and the abstract semantics, and rely on an *abstract execution engine* to execute the program using the abstract semantics during the analysis. To handle pointers, we use abstract values to track block-size dependence of the *address* of location represented by a pointer. Hence, when a pointer is dereferenced, we conservatively return $\mathsf{c}_{\mathsf{ind}}/\mathsf{b}_{\mathsf{ind}}$ if the pointer is constant, and $\mathsf{c}_{\mathsf{bsize}}/\mathsf{b}_{\mathsf{bsize}}$ if the pointer is not a constant. We only update the value of a pointer variable on pointer assignment and pointer updates through indexing. Structures are represented similar to arrays in LLVM and hence no special semantics are necessary.

We represent multipliers in the abstract values as follows. LLVM exposes each variable in the program as a unique `Value*` pointer. We use this pointer to represent the multiplier and compare it against other pointers. Since LLVM uses the SSA form, the pointer corresponds to a unique definition and the value for the variable is not updated after it is first defined. Note that the program variables which are accessed via load/store instructions, do not appear as operands in regular arithmetic or boolean operations, and vice-versa. Hence, such variables are never used as multipliers in the abstract domain and the value for the multipliers is never updated through indirect store operations.

Correctness. We show the correctness of our analysis. The analysis preserves the abstraction and ensures that each variable gets an abstract value

$\mathsf{c}_{\mathsf{ind}}/\mathsf{b}_{\mathsf{ind}}$ only if the value is truly block-size independent, *i.e.* the assigned value and the path-predicate are block-size independent. Further, each write to global variables is guarded by a check for block-size independence. Therefore, if the analysis does not report any block-size dependent writes, the updates to the global memory are always block-size independent, and the global state at the end of each thread's execution must also be block-size independent. This implies the program is thread-local block-size independent, and hence, we conclude the following theorem.

Theorem 2. *A synchronization-free GPU program* $\langle d, V_L, V_S, V_G, C, K\rangle$ *is* block-size independent*, if the analysis reports the program to be block-size independent.*

4 Evaluation

We have implemented the block-size independence analysis in LLVM 7.0, a popular open-source compiler framework, and evaluate it on the Nvidia CUDA SDK 8.0 sample programs. The SDK consists of 62 applications, out of which 28 benchmarks rely on texture memory fetches and the Thrust library and could not be compiled with LLVM. Hence, we analyze the remaining 34 benchmarks. For each benchmark, we analyze *global* kernels which are entry-points into the call-graph and are invoked directly from CPU code. For each global kernel, the analysis reports whether the kernel is block-size independent (BSI), and if not, the potential block-size dependent accesses in the kernel. We run the analysis on an Amazon EC2 machine with 4-core Intel Xeon 2.3GHz CPU and 16GB memory running Ubuntu 16.04 LTS (OS).

How many BSI kernels are found by the analysis? Table 1 shows the results for the analysis. The graph shows the the total number of global kernels and the number of BSI kernels reported by our analysis. Note that in few of the benchmarks, the global kernels are instantiations of templated kernels. The global kernels have similar functionality, and hence, the numbers are slightly bloated. For example, in benchmarks "reduction", "threadFenceReduction", and "alignedTypes", the total number of kernels is 132, 40 and 16, though these are instantiations of 7, 2 and 1 templated kernels, respectively. Yet, the analysis is able to verify a large number of kernels as BSI. It finds 35 BSI kernels in 11 benchmarks, and runs in a few seconds for most benchmarks, rarely taking more than a minute.

Are there truly non-BSI kernels? We manually investigated the benchmarks and found a few non-BSI kernels. These kernels asymmetrically distribute computation between blocks and threads, and hence, are block-size dependent. For example, benchmarks "binomialOptions" and "MonteCarloMultiGPU" allocate an 'option' per block while the threads collaborate to compute the value for the option. Similarly, "scalarProd" allocates a vector-pair per block while the threads multiply and add individual elements to get the scalar product.

What class of kernels could not be verified? We could not verify block-size independence for kernels where shared memory and thread-synchronization

Benchmark	# Kernels	# BSI	Benchmark	# Kernels	# BSI
Mandelbrot	6	0	concurrentKernels	2	0
simpleGL	1	0	eigenValues	4	0
convolutionSeparable	2	0	fastWalshTransform	3	2
cudaDecodeGL	2	2	FDTD3dGPU	1	0
dwtHaar1D	2	0	interval	1	0
histogram	4	0	mergeSort	7	3
recursiveGaussian	3	2	newDelete	14	4
simpleCUDA2GL	2	2	reduction	132	0
binomialOptions	1	0	scalarProd	1	0
BlackScholes	1	0	scan	3	0
MonteCarloMultiGPU	2	0	shfl_scan	4	0
quasiRandomGenerator	2	2	SimpleHyperQ	3	0
SobolQRNG	1	1	sortingNetworks	6	0
nbody	2	0	StreamPriorities	1	1
oceanFFT	3	2	threadFenceReduction	40	0
alignedTypes	12	12	threadMigration	1	0
cdpLUDecomposition	2	0	transpose	8	0

Table 1: Results of BSI analysis for Nvidia CUDA SDK 8.0 samples. # Kernels represents the total number of global kernels. # BSI represents the number of these kernels that are block-size independent.

were used to intricately share data between threads within a block. A common scenario was a parallel reduction operation such as summing elements. The block-size was hard-coded via `#define` constants for few of the kernels, which prevented verification. We observed an interesting pattern in benchmarks "dwtHaar1D" and "reduction" where each thread operated on two locations in a global array: $(2\mathsf{bid}.\mathsf{bdim} + \mathsf{tid})$ and $(2\mathsf{bid}.\mathsf{bdim} + \mathsf{bdim} + \mathsf{tid})$. The locations individually are block-size dependent. However, cumulatively, the threads operate on all elements, which makes the operation block-size independent. Finally, we could not verify kernels in "simpleGL", "oceanFFT" and "interval" to be BSI, because library calls containing inline assembly calls and addition between integers and booleans were inlined into the kernels, which were falsely reported block-size dependent.

Does tuning block-size for BSI kernels improve performance? We experimented with benchmark "SobolQRNG" to gauge performance improvement via block-size tuning. The benchmark originally used shared memory to cache global constants and was reported non-BSI by our analysis. The block-size was set to 64 threads/block and produced 18.8 Gsamples/s (baseline) on an Nvidia GTX Titan X GPU. We removed caching to obtain a BSI version. Here for 64 threads/block, we lost performance by 40% (11.6 Gsamples/s), but then for 256 threads/block, we regained performance with an improvement of 9% over the baseline (20.5 Gsamples/s). Our analysis helped tune block-size to gain performance while ensuring correctness, unlike the other optimization.

How many kernels could be easily fixed to become BSI? We fixed 7 kernels to be BSI with our analysis (included in the 35 BSI kernels found by the analysis). In "quasiRandomGenerator" and "fastWalshTransform", the number of blocks for the second grid dimension was set to 1, and thus bid_1 was always set to 0 and dropped from the computation for gid_1. In "cudaDecodeGL", gid was computed as $(\mathsf{bdim}).(\mathsf{bid} << 1) + (\mathsf{tid} << 1)$, where the '<<' operator was not supported by our analysis. Finally, in "quasiRandomGenerator", gid was computed as $(\text{mul}(\mathsf{bid}, \mathsf{bdim}) + \mathsf{tid})$, where the 'mul' method was not supported.

5 Related Work

Auto-tuning. A rich body of work exists on automatically tuning GPU applications for specific hardware configurations. Broadly, there are three types of auto-tuning: *empirical tuning* [28, 13, 20, 16, 23, 27], where different program variants are executed and the best variant is identified via exhaustive search or a hill-climbing approach; *model-based tuning* [4, 5], where a hand-crafted model is used to select the best program variant; and *predictive model-based tuning* [26, 13, 9, 14, 1], where a predictive model trained via machine learning techniques like decision trees is used to select the best program variant. All these approaches either automatically generate the final GPU program, or transform an existing program to generate the tuned program. A few of these works tune block-size directly [13, 14, 1, 27], but do not verify correctness of the transformation. A few are domain-specific [28, 20, 5, 16, 23], often using programs written in a domain-specific languages instead of CUDA and OpenCL. Finally, many recent works focus on data-layout optimization [26, 9] and data placement [4]. These works segregate specification of data-layout and data-placement from the actual program by hiding it under a data-abstraction layer. Hence, only the spec for data-layout and placement is modified during auto-tuning and the program remains unchanged. This localizes any errors to the implementation of data-layout specifications, which ensures greater correctness. Tuning block-size is, however, essential to utilize resources on GPUs effectively, and our work on validating block-size independence can enable robust auto-tuning for block-size transformation.

GPU Verification. Several systems exist for verification of GPU programs. GKLEE [12] and KLEE-CL [6] extend KLEE, a popular symbolic execution engine, to verify GPU programs against data-races and barrier divergence. Due to the presence of a large number of threads, these tools do not scale to large programs. GPUVerify [2] and PUG [11] improve upon GKLEE and KLEE-CL, by using *symbolic threads* and SMT-based verification to identify data-races. The underlying SMT solvers have trouble scaling to very large formulae as well. Finally, Leung et al. [10] present an approach where they analyze programs for *input-independence*, verify safety properties of input-independent programs for a small set of inputs and then generalize results to all other inputs. The analysis to verify input-independence is similar to ours, except it tracks the flow of input variables instead of the block-size dependent constants.

Abstract Interpretation + Symbolic Execution. A few works, similar to our work, use symbolic constants to improve precision of an abstract domain, while retaining the scalability of the analysis. Sankaranarayanan et al. [22] and Venet [25] extend the Interval domain with symbolic ranges, where the upper and lower bounds of an interval are a linear combination of symbolic constants representing program variables. Miné [15] presents two generic techniques: *linearization*, which instantiates symbolic variables with abstract constants to obtain a linear expression in symbolic variables, and *symbolic constant propagation*, which propagates symbolic constants across expressions to gain precision.

6 Conclusion

The paper formalizes block-size independence for GPU programs and presents an inter-procedural analysis to verify block-size independence for synchronization-free programs. The analysis relies on tracking the flow of block-size dependent values via an abstraction that combines symbolic multipliers with abstract constants representing different dependencies on block-size. It is very efficient and finds a considerable number of block-size independent global kernels in Nvidia CUDA SDK.

In future, we would like to extend the analysis to GPU programs with restricted synchronization between threads, by either transforming these programs into synchronization-free programs or ensuring that the execution of each thread is independent of the set of threads it synchronizes with, and then the present analysis would suffice to prove block-size independence of the programs.

We would like to thank the anonymous reviewers and our shepherd Sylvie Putot for their valuable feedback. We would also like to thank NSF award XPS-1337174 and hardware donations from Nvidia for supporting this research.

References

1. Bergstra, J., Pinto, N., Cox, D.: Machine learning for predictive auto-tuning with boosted regression trees. In: 2012 Innovative Parallel Computing (InPar). pp. 1–9 (May 2012)
2. Betts, A., Chong, N., Donaldson, A., Qadeer, S., Thomson, P.: GPUVerify: A verifier for GPU kernels. SIGPLAN Not. 47(10), 113–132 (Oct 2012), http://doi.acm.org/10.1145/2398857.2384625
3. Boyer, R.S., Elspas, B., Levitt, K.N.: SELECT – a formal system for testing and debugging programs by symbolic execution. In: Proceedings of the International Conference on Reliable Software. pp. 234–245. ACM, New York, NY, USA (1975), http://doi.acm.org/10.1145/800027.808445
4. Chen, G., Wu, B., Li, D., Shen, X.: PORPLE: 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

5. Choi, J.W., Singh, A., Vuduc, R.W.: Model-driven autotuning of sparse matrix-vector multiply on GPUs. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 115–126. PPoPP '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1693453.1693471
6. Collingbourne, P., Cadar, C., Kelly, P.H.J.: Symbolic testing of OpenCL code. In: Haifa Verification Conference (HVC 2011) (1 2011)
7. 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
8. King, J.C.: A new approach to program testing. In: Proceedings of the International Conference on Reliable Software. pp. 228–233. ACM, New York, NY, USA (1975), http://doi.acm.org/10.1145/800027.808444
9. Kofler, K., Cosenza, B., Fahringer, T.: Automatic data layout optimizations for GPUs. In: Träff, J.L., Hunold, S., Versaci, F. (eds.) Euro-Par 2015: Parallel Processing. pp. 263–274. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)
10. Leung, A., Gupta, M., Agarwal, Y., Gupta, R., Jhala, R., Lerner, S.: Verifying GPU kernels by test amplification. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 383–394. PLDI '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2254064.2254110
11. 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
12. 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
13. Liu, Y., Zhang, E.Z., Shen, X.: A cross-input adaptive framework for GPU program optimizations. In: 2009 IEEE International Symposium on Parallel Distributed Processing. pp. 1–10 (May 2009)
14. Magni, A., Dubach, C., O'Boyle, M.: Automatic optimization of thread-coarsening for graphics processors. In: Proceedings of the 23rd International Conference on Parallel Architectures and Compilation. pp. 455–466. PACT '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2628071.2628087
15. Miné, A.: Symbolic methods to enhance the precision of numerical abstract domains. In: Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation. pp. 348–363. VMCAI'06, Springer-Verlag, Berlin, Heidelberg (2006), http://dx.doi.org/10.1007/11609773_23
16. Monakov, A., Lokhmotov, A., Avetisyan, A.: Automatically tuning sparse matrix-vector multiplication for GPU architectures. In: Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers. pp. 111–125. HiPEAC'10, Springer-Verlag, Berlin, Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-11515-8_10
17. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. Queue 6(2), 40–53 (Mar 2008), http://doi.acm.org/10.1145/1365490.1365500

18. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer Publishing Company, Incorporated (2010)
19. Nvidia: Nvidia CUDA SDK, https://developer.nvidia.com/cuda-code-samples/
20. Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S.: Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 519–530. PLDI '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2491956.2462176
21. Ryoo, S., Rodrigues, C.I., Stone, S.S., Baghsorkhi, S.S., Ueng, S.Z., Stratton, J.A., Hwu, W.m.W.: Program optimization space pruning for a multithreaded gpu. In: Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization. pp. 195–204. CGO '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1356058.1356084
22. Sankaranarayanan, S., Ivančić, F., Gupta, A.: Program analysis using symbolic ranges. In: Proceedings of the 14th International Conference on Static Analysis. pp. 366–383. SAS'07, Springer-Verlag, Berlin, Heidelberg (2007), http://dl.acm.org/citation.cfm?id=2391451.2391476
23. Sørensen, H.H.B.: Auto-tuning dense vector and matrix-vector operations for Fermi GPUs. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) Parallel Processing and Applied Mathematics. pp. 619–629. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
24. Stone, J.E., Gohara, D., Shi, G.: OpenCL: A parallel programming standard for heterogeneous computing systems. IEEE Des. Test 12(3), 66–73 (May 2010), http://dx.doi.org/10.1109/MCSE.2010.69
25. Venet, A.J.: The gauge domain: Scalable analysis of linear inequality invariants. In: Proceedings of the 24th International Conference on Computer Aided Verification. pp. 139–154. CAV'12, Springer-Verlag, Berlin, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-31424-7_15
26. Weber, N., Goesele, M.: MATOG: Array layout auto-tuning for CUDA. ACM Trans. Archit. Code Optim. 14(3), 28:1–28:26 (Aug 2017), http://doi.acm.org/10.1145/3106341
27. Yang, Y., Xiang, P., Kong, J., Mantor, M., Zhou, H.: A unified optimizing compiler framework for different GPGPU architectures. ACM Trans. Archit. Code Optim. 9(2), 9:1–9:33 (Jun 2012), http://doi.acm.org/10.1145/2207222.2207225
28. Zhang, Y., Mueller, F.: Auto-generation and auto-tuning of 3D stencil codes on GPU clusters. In: Proceedings of the Tenth International Symposium on Code Generation and Optimization. pp. 155–164. CGO '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2259016.2259037