Warp-Level CFG Construction for GPU Kernel WCET Analysis

Louison Jeanmougin
IRIT - Univ. Toulouse 3 - CNRS, France

Pascal Sotin
IRIT - Univ. Toulouse 2 - CNRS, France

Christine Rochange
IRIT - Univ. Toulouse 3 - CNRS, France

Thomas Carle
IRIT - Univ. Toulouse 3 - CNRS, France

Abstract

We present an abstract interpretation technique to automatically build a Control Flow Graph (CFG) representation of the execution of a GPU kernel. GPUs implement an inherently parallel execution model, in which threads are grouped within so-called warps that execute in lockstep. This execution model enables the representation of the execution of the threads of a warp as a single CFG. However, thread divergence may appear within a warp and its effect must be captured explicitly within the CFG. Our method builds the CFG of a warp by applying abstract interpretation on the assembly (Nvidia SASS) code of a kernel, and by maintaining an abstract representation of which threads within the warp agree on which values. This allows the method to detect precisely the points in the program where thread divergence may occur, and avoid spurious reactivation edges in the CFG. We apply our technique on benchmark kernels as a proof-of-concept, and generate IPET systems using the resulting CFGs.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Theory of computation → Abstraction

Keywords and phrases Graphical Processing Unit (GPU), Control Flow Graphs (CFG), Worst-Case Execution Time (WCET), Program analysis

Digital Object Identifier 10.4230/OASIcs.WCET.2023.1

Funding *Thomas Carle*: This work was supported by a grant overseen by the French National Research Agency (ANR) as part of the MeSCAliNe (ANR-21-CE25-0012) project.

1 Introduction

The ever-growing need for computation power begs the question of adopting hardware accelerators in real-time embedded systems. In particular, Graphical Processing Units (GPUs) have gained traction as they combine massive parallelism and versatility. However, their adoption for safety-critical real-time systems requires the ability to derive safe Worst-Case Execution Time (WCET) bounds for the programs accelerated by GPUs. Traditional static WCET analysis targets the execution of a sequential thread running in isolation on a CPU core. In practice, the embedded program is modelled using a Control Flow Graph (CFG) that captures all the possible execution paths of the program in a condensed representation. Abstract interpretation techniques can be applied on this graph to determine properties of the program execution (e.g. loop bounds, infeasible paths, cache behavior). Each node of the CFG corresponds to a sequence of instructions of the program whose worst-case execution duration is derived using a model of the target hardware (in fully static methods) or using measurements (in hybrid methods). The CFG is ultimately used in the Implicit

Path Enumeration Technique (IPET) in order to generate an Integer Linear Program (ILP) system that captures the possible execution paths and combines them with the worst-case execution duration of the nodes of the CFG. The solution of this ILP system is the WCET bound for the considered program. Additional techniques can be used to account for the effect of concurrent threads running on the same core [1] or on different cores of a multi-core System-on-Chip [8, 11].

In order to handle thousands of threads in parallel, GPUs implement a complex execution model in which the threads executing a program are hierarchically subdivided into groups called thread blocks and warps. Thread blocks are dynamically distributed among the Streaming Multiprocessors (SM) composing a GPU following occupation rules defined in the GPU drivers [2, 12]. Within each block, the threads are also grouped into warps. Threads within a warp execute in lockstep: at each execution cycle, each multiprocessor elects a warp for execution, and all the threads within the elected warp execute the same instruction. This execution model is known as Single Instruction Multiple Threads (SIMT). Since all the threads within a warp execute the same instruction at the same time, it seems natural to derive the worst-case execution time of a warp in isolation, using a single CFG, and following the classical WCET analysis workflow. From this information, additional analyses can then be developed and applied to combine multiple warps running on the same GPU and derive a WCET for the complete application.

However, the SIMT execution model is subject to a phenomenon called thread divergence that impacts the control flow, and ultimately the execution time, by serializing the execution of the different branches of conditional branch statements when the threads within a warp do not agree on the value of the condition. For each thread taken separately this has no impact on the control flow, but at warp level, this serialization mechanism creates additional transitions in the control flow that must be accounted for in the warp-level CFG.

In this paper, we present an abstract interpretation technique that builds an accurate warp-level CFG directly from the assembly code (Nvidia SASS) of the application. This technique models the semantics of the SIMT execution model, and is able to determine a subset of the conditional branch instructions for which the threads are statically guaranteed to agree on the execution condition.

Building the CFG by abstract execution of the machine code in a CPU context is known to bring the following benefits [3]:

1. independence from the source language and the compilation process;
2. more accurate control graph and abstract values thanks to interleaving of value analysis and graph construction.

In the context of GPU timing analysis, this approach seems even more natural because:

- the source code describes the behavior of *each* thread, while the machine code describes the behavior of a warp;
- the subsequent WCET analysis is performed on the machine code[1]

The paper is organized as follows. In Section 2, we present the details of the SIMT semantics. We then present our CFG construction method in Section 3 and evaluate it in Section 4. We present the related work on the topic of static WCET analysis of GPU kernels in Section 5 and we conclude in Section 6.

[1] We leave aside the problem of transferring the loop bound information from source to assembly code.

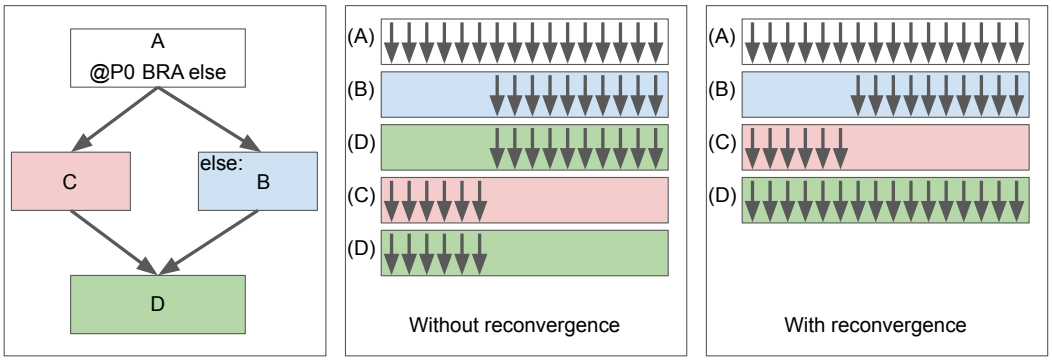

Figure 1 Thread divergence example.

2 SIMT execution semantics

As mentioned earlier, GPUs implement a particular execution model called SIMT. At the highest level, the main CPU program calls a GPU function (called a kernel) that is executed by a specified number of threads. These threads are organized into blocks that are dynamically dispatched to the SMs composing the GPU. Within a block, threads are divided in groups of 32 (or 16 depending on the GPU architecture) called warps. Inside an SM, warp schedulers are responsible for selecting a warp to execute at each execution cycle. Threads within a warp execute in lockstep: whenever a warp is elected for execution, all the threads that compose it execute the same instruction. This greatly simplifies the logic, as all threads within a warp can be seen as sharing their program counter (PC). However this execution model can be problematic when the threads execute conditional branches: in certain situations, called thread divergence, threads within a warp may not agree on the value of the execution condition of a branch, and thus on the next value of their shared PC. This is handled by executing the branch with the threads that find the execution condition true, while masking the others, and then executing the fallback code with the other threads only. This mechanism is illustrated in Figure 1. On the left, the figure displays a simple CFG with a conditional branch: at the end of block A, the control flows towards B or C, and then reaches D regardless. At the end of block D, each thread executes the `EXIT` instruction that signals the end of execution for the thread. A possible execution for a warp is given in the middle. At the beginning, all threads within the warp[2] execute block A. Then the ten rightmost threads execute block B followed by block D, until they reach the end of the program. When the execution is over for these threads, the six leftmost threads execute block C, followed by block D. At this point, all threads have finished executing the kernel.

This serialization allows the correct execution of a kernel, but is not efficient, as some parts of the code (located after the if-then-else) are executed multiple times in sequence (e.g. block D in our example). A reconvergence mechanism is implemented to reduce the subsequent loss of performance: the compiler automatically detects areas in the code where thread divergence may occur (around conditional branch instructions) and adds special purpose instructions in the assembly code to re-synchronize the threads that have diverged. This is illustrated in the right part of Figure 1. As before, at the beginning all threads

[2] For space reasons we only depict 16 threads within the warp.

execute block A. Then the ten rightmost threads execute block B and are suspended, as they reach a reconvergence instruction at the end of the block. The six leftmost threads execute block C, and when they are done, all threads reconverge before executing block D.

The SASS instruction set (up to the Maxwell/Pascal ISA at least) contains two pairs of such instructions: `SSY`/`SYNC` and `PBK`/`BRK`. In each pair, the first instruction is used to signal a potential incoming divergence to the hardware, while the second instruction is used to synchronize diverging threads, thus forcing their reconvergence. In order to support nested conditional branches, an activation stack stores the necessary information: each entry in the stack is composed of a mask representing the active threads of the entry, and of the next PC value for these threads. The entry at the top of the stack always represents the currently active threads and the PC of the next instruction to execute. Additionally, each entry of the stack is typed in order to handle intertwined loop, if-then-else and break constructs:

- an entry is of `NIL` type if it does not correspond to a reconvergence point. The top of the stack is always of `NIL` type.
- an entry is of type `SYNC` if it was inserted using the `SSY` instruction. It usually is used to reconverge after a conditional branch due to a if-then-else or a loop construct.
- an entry is of type `BRK` if it was inserted using the `PBK` instruction. It usually is used to reconverge after a conditional break statement.

The `SSY` and `PBK` instructions contain the address of the instruction at which the corresponding reconvergence must occur.

The stack is maintained using the following rules:

- when a warp is mapped to an SM, a stack is allocated. It is initially composed of a single `NIL` entry with all threads active in the mask and the next PC corresponding to the start of the kernel code.
- when a warp reaches a `SSY @reconv_addr` (resp. `PBK @reconv_addr`) instruction, the next PC of the top entry becomes `reconv_addr`, and the entry becomes typed as `SYNC` (resp. `BRK`). This entry will be used when the threads reconverge. A new `NIL` entry is then pushed on the stack. This entry has the same thread mask as the previous top entry (i.e. all currently active threads remain active), and its next PC is set to the current PC + 8 (i.e. the next instruction in memory for 64 bit instructions). The `SSY` (resp. `PBK`) instruction prepares the stack for a potential divergence due to a future conditional branch instruction, but is not by itself a source of divergence.
- when a warp reaches a divergent conditional branch `BRA @addr`, the actual divergence must be accounted for in the stack. The top entry next PC is set to its current value + 8 (i.e. the address of the fallback code), and its mask is updated to contain only the active threads that do not take the branch. This entry will be used later to execute the fallback code. A new `NIL` entry is pushed to the stack. Its next PC is set to `addr` and its mask is composed of the active threads that take the branch.
- when a warp reaches a `SYNC` (resp. `BRK`) instruction, the active threads are removed from the mask of all entries in the stack, from the top and until a `SYNC` (resp. `BRK`) entry is reached. Each time a mask is modified, its corresponding entry is popped from the stack if the modified mask no longer contains any thread. In practice, the `SYNC` (resp. `BRK`) instruction suspends the execution of the currently active threads until a reconvergence point is reached by all the threads that must reconverge.
- when a warp reaches an `EXIT` instruction (i.e. the active threads reach the end of the kernel), all active warps that execute the `EXIT` are removed from the mask of all the entries in the stack. Once again, if a mask becomes empty doing so, its entry is removed from the stack.
- whenever an entry becomes the top entry, its type becomes `NIL` regardless of what it was before.

This set of rules is implemented in the hardware, and as the `SSY/SYNC`, `PBK/BRK` and `EXIT` instructions are automatically inserted by the compiler, the process of handling thread divergence when it occurs is transparent to the programmer. On the other hand, our static analyses follow closely this execution model in order to accurately account for its effect. In particular, to the best of our knowledge, it is the first time that a static analysis is performed at the granularity level of the assembly language for a GPU kernel and that the divergence/reconvergence instructions are taken into account. This is particularly important as in our experiments, we have encountered situations in which the compiler does not insert reconvergence instructions on the first post-dominating node after a conditional branch.

In the next section, we present our abstract interpretation method to build a warp-level CFG from the SASS code of a GPU kernel.

3 Abstract interpretation

In the manner of Reps and al. [3], we perform the CFG construction at the machine code level and we interleave exploration of the control flow with value analysis.

We construct the graph by starting at the entry point of the program with an initial abstract state representing the possible initial concrete states. We execute the instructions of the program on the abstract states and discover successors states. We do so until a fixpoint is reached and no more new states are discovered. An abstract state contains information on:

- the current program counter;
- the pending activation stack (see Section 3.1);
- several remarkable groups of threads (see Section 3.2);
- for each group, the registers on which these threads agree (see Section 3.3).

After presenting the base domains in the following subsections, we detail our abstract states in Section 3.4. For each domain, we provide a formal description of the concretization function γ, that associates a value in an abstract domain $D^\sharp$ to a set of concrete values, and a description of its essential operations.

The notation $A \to B$ denotes a total function from A to B; $A \rightharpoonup B$ denotes a partial function from A to B; $\mathcal{P}(A)$ denotes a subset of A.

3.1 Activation stack abstract domain

We represent the possible configurations of the activation stack by a graph in which the nodes are composed of a control point and an optional tag (`SYNC`, `BRK` or *none*), plus two special nodes : *top* and *bot*. In such a graph, each path from *top* to *bot* represents the contents of a stack in a possible configuration. The stack concretization function γ_{Stack} takes as input a graph and returns the set of corresponding stack configurations.

$$\gamma_{\text{Stack}} : D^\sharp_{\text{Stack}} \to \mathcal{P}(\text{List}(Act^\sharp)) \qquad \text{with } Act^\sharp \stackrel{\text{def}}{=} PC \times Tag$$

Operations

Most operations in this abstract domain are simple adaptations of classical operations on stacks. We detail the pop and filter operations.

Popping

While pushing an element on an abstract stack simply gives an abstract stack, popping the topmost element may yield several possible popped values and leave distinct remainders. It might also happen that the stack can be empty; *none* is then part of the result.

$$pop^\sharp : D^\sharp_{\text{Stack}} \to \mathcal{P}(Act^\sharp \times D^\sharp_{\text{Stack}})$$

This operation is called after the active threads have been halted by an `EXIT`, `SYNC` or `BRK`. The execution may continue at any of the control point that can be popped. If the stack may be empty, this indicates that the program can halt here.

Filtering

When an operation halts the active threads, it removes them from the active mask, but also from all the masks in the stack, up to the bottom in case of an `EXIT` and up to the corresponding tag in case of a `SYNC` or `BRK`.

Our representations of the stack do not embed the masks[3]. However, we need to take into account that stages of the activation stack might have been removed after that their activation mask attained the zero vector.

We thus equipped our abstract stack domain with a filtering operation that processes the stages from the top of the stack up to an optional tag and for each stage take into account that its mask can/cannot/must reach zero. The latter information is taken as a parameter and is provided by the thread group abstract domain (Sec. 3.2).

$$filter^\sharp : D^\sharp_{\text{Stack}} \times Tag \times (Act^\sharp \to \{\texttt{keep}, \texttt{drop}, \texttt{any}\}) \quad \to \quad D^\sharp_{\text{Stack}}$$

3.2 Thread group abstract domain

In order to conduct a precise analysis, we sometimes need to retain the relations between certain groups[4] of threads. We thus introduced an abstract domain that is able to remember relations like $A \subseteq B$, $A = B$ or $A \cap B = \emptyset$.

The domain does not keep track of the concrete threads present in a group, just the relations between these groups. The group names are stored in the abstract value. In our case, we use the special group `active` to denote the threads currently executing and one group per activation node in the stack graph to denote either its associated mask or an upper bound on it when in a cycle. This set of groups is denoted by G.

$$\gamma_{\text{Group}} : D^\sharp_{\text{Group}} \quad \to \quad \mathcal{P}(G \rightharpoonup \mathcal{P}(Threads))$$

We implement this domain by storing all the intersections of groups or complements of groups that must be empty. The constraint $A \cap B = \emptyset$ is stored as is, the constraint $A \subseteq B$ is stored as $A \cap \bar{B} = \emptyset$ and the constraint $A = B$ is stored as $A \subseteq B \wedge B \subseteq A$. The conjunction of constraints is an exact operation[5] in this domain.

[3] Putting the masks or abstraction thereof in the abstract stack would lead to very large graphs.
[4] In this specific context, we use the term *group*, but it can be read as *set*.
[5] An *exact operation* is an operation of the abstract domain that does not result in an over-approximation.

3.3 Agreement abstract domain

Thread divergence can occur when a conditional branch instruction is executed. However, in many situations all the threads of the warp agree on the predicate, by program design.

In order to determine if a divergence may occur or not, we created a new abstract domain that keeps track for a given group of threads of the registers on which these threads agree. We just store the name of these registers, not the values they contain. The beauty of this analysis is that we do not need to know the precise behaviour of each instruction but only what it reads, what it writes, and be sure that it is deterministic.

For example, if the instruction is `IADD R2, R4, R2;` we can tell that if a group of threads agree on `R2` and `R4`, they will all write the same value in `R2` and thus keep agreeing on these registers.

The basic version of this domain captures the identical registers in a given group of threads.

$$\gamma_{\text{Agree}} : (\mathcal{P}(\mathit{Thread}) \times \mathcal{P}(\mathit{Reg})) \quad \to \quad \mathcal{P}(\mathit{Mem})$$
$$\gamma_{\text{Agree}}(T, R) = \{m \in \mathit{Mem} \mid \forall t_1, t_2 \in T, \forall r \in R, \text{read}(m, t_1, r) = \text{read}(m, t_2, r)\}$$

Such abstract value is concretized as the set of memories such that any two threads in the group that read the same register on which the agreement was established, read the same value. Values in *Mem* describe both the registers and the DRAM memories of the GPU.

This domain is then lifted to handle several groups of threads, identified in Section 3.2.

$$D^\sharp_{\text{GrAgr}} \stackrel{\text{def}}{=} G \rightharpoonup \mathcal{P}(\mathit{Reg})$$
$$\gamma_{\text{GrAgr}} : D^\sharp_{\text{GrAgr}} \quad \to \quad \mathcal{P}((G \rightharpoonup \mathcal{P}(\mathit{Threads})) \times \mathit{Mem})$$
$$\gamma_{\text{GrAgr}}(f) \stackrel{\text{def}}{=} \{\langle g, m\rangle \in (G \rightharpoonup \mathcal{P}(\mathit{Threads})) \times \mathit{Mem} \mid \text{dom}(g) = \text{dom}(f) \quad \wedge \quad \forall x \in \text{dom}(f), m \in \gamma_{\text{Agree}}(g(x), f(x))\}$$

When we process an instruction computing data, we update the information tied to each group in the following manner:

- If the group is equal to `active`, we perform a strong update. If the threads agree on the arguments, they gain or preserve the agreement on the result. If the threads disagree on at least one argument, agreement on the result is lost.
- If the group is not equal to `active` but intersecting it, we perform a weak update. It means that only a part of the group performs the instruction. Agreement on the result is thus lost.
- If the group is disjoint from `active`, we do not modify its agreements.

3.4 Warp state abstract domain

As announced on page 5, a state is made of the current program counter, the pending activation stack, a set of remarkable groups of threads and agreement information on these groups.

$$D^\sharp_{\text{Warp}} \stackrel{\text{def}}{=} \mathit{PC} \times D^\sharp_{\text{Stack}} \times D^\sharp_{\text{Group}} \times D^\sharp_{\text{GrAgr}}$$
$$\gamma_{\text{Warp}} : D^\sharp_{\text{Warp}} \quad \to \quad \mathcal{P}(\text{List}((\mathit{PC} \times \mathit{Tag}) \times \mathcal{P}(\mathit{Thread})\}) \quad \times \quad \mathit{Mem})$$
$$\gamma_{\text{Warp}}(p, s^\sharp, g^\sharp, a^\sharp) \stackrel{\text{def}}{=} \{\langle\langle\langle p, \mathit{none}\rangle, g(\texttt{active})\rangle.s', m\rangle \mid \langle g, m\rangle \in \gamma_{\text{GrAgr}}(a^\sharp) \quad \wedge \quad g \in \gamma_{\text{Group}}(g^\sharp) \wedge \exists s \in \gamma_{\text{Stack}}(s^\sharp), \forall i, s'[i] = \langle s[i], g(s[i])\rangle\}$$

Processing an instruction

We define the abstract treatment of an instruction by separating the concerns of processing an instruction unconditionally (memo + operands) from the treatment of the condition. This organization of the abstract semantics avoids to consider for each kind of instruction a tedious and error-prone study of the interference between the instruction and the presence of a condition.

Managing the condition

Processing an instruction with its condition may produce several abstract values, that we do not wish to join immediately into a single one. An informal algorithm is given in Algo. 1 and it uses the **produce** keyword to signal one or several results (as `yield` in Python).

Algorithm 1 Successors of an abstract state: condition management.

Input : An initial abstract state st_{ini}
Output : The production of one or more successor state

$PC_{\text{cur}} \leftarrow PC(st_{\text{ini}})$;
$i \leftarrow$ instruction at PC_{cur};
$st_{\text{all}} \leftarrow$ process unconditionally instruction i in state st_{ini};
if *st_{all} has no more active threads* **then**
  produce all pop from st_{all};
else
  produce st_{all};
if *i is conditional* **then**
  produce st_{ini} with $PC = PC_{\text{cur}} + 8$;
  if *active threads might disagree on cond(i) in st_{ini}* **then**
    $st_{\text{part}} \leftarrow st_{\text{ini}}$ with a group `skip` separated from `active`;
    $st_{\text{some}} \leftarrow$ process unconditionally instruction i in state st_{part};
    if *st_{some} has no more active threads* **then**
      produce st_{some} with group `skip` renamed as `active`;
    else if $PC(st_{some}) = PC_{cur} + 8$ **then**
      produce st_{some} with group `skip` folded into `active`;
    else
      produce st_{some} with group `skip` pushed as $\langle PC_{\text{cur}} + 8, none \rangle$;

Processing unconditional instructions

1. Branching (`BRA`) replaces the current PC with the target of the instruction. Non-branching instructions increment the PC *in addition to their effect*.
2. Data-processing instructions (eg. `IADD` or `LDC`) modify the agreement component of the abstract value, as presented at the end of Section 3.3.
3. Reconvergence preparation instructions (`SSY` and `PBK`) push their target on the pending activation stack with a tag corresponding to the kind of synchronization.
4. Reconvergence instructions (`SYNC` and `BRK`) split the abstract stack on the first occurrence of the corresponding tag. The upper part is then filtered from the activation stages that can or must have their mask reduced to zero when we halt the threads in `active`.
5. The `EXIT` instruction filters the `active` threads from the whole stack.

In the next section we describe the results that we obtained using our method.

4 Evaluation

4.1 Experimentation on the Rodinia Benchmark

To the best of our knowledge, no GPU kernel benchmark dedicated to embedded or real-time systems has been released yet. To evaluate our analysis method, we thus used kernels extracted from the Rodinia [7] benchmark. Since our analysis does not support calls to kernels originated from another kernel (we are currently working on understanding the relationship between the call stack and the reconvergence stack), our evaluation was performed on 37 out of the 57 kernels composing Rodinia. For each of them, our prototype was able to generate a CFG, and from each CFG we generated an ILP system following the IPET method. Developing a loop-bound analysis for warp-level CFGs is part of future work, so for now we manually provided the loop bounds, and arbitrarily set each loop bound to 10 iterations (or a power of ten for nested loops). Loop reconstruction was done using [6]. Additionally, we arbitrarily set each instruction duration to 1 cycle in order to obtain durations for the blocks of the produced CFGs. Our objective was not to derive a real WCET for these kernels but to prove the feasibility of deriving a warp-level CFG and to use it in a standard IPET workflow.

Overall, most of the analyzed kernels have a very limited number of arcs modelling a possible divergence (34 of them have 4 or less of these arcs), which is coherent since most of these kernels are pretty simple, and do not feature if-then-else constructs. Interestingly, the 3 kernels with a slightly more complex control flow (switch-case and if-then-else constructs) have 39, 19 and 12 of them (respectively in the mummergpuKernel, printKernel and reduce kernels). This means that the kind of analysis that we propose is necessary in order to support even relatively simple kernels. For each of them we were able to compute a WCET, which shows that the generated CFGs are compatible with the IPET method. The CFG reconstruction took up to 1.3 seconds; in 75% of the benches, it took less than 90 ms.

In the future, we plan on looking at other benchmarks to try to find more complex kernels, or to develop our own benchmark with kernels that can be interesting for analysis or representative of embedded GPU kernels.

4.2 Evaluation of the abstract domain

In this subsection we give some results on the relative importance of the component of the abstract domain $D^\sharp_{\text{Warp}}$ presented in Section 3. We deactivate some parts of the abstract value and observe the fraction of the benchmark programs of Section 4.1 that see their WCET severely degraded. The results are the following:

Domain modification	WCET est. $\times 10$ or more
Limitation to acyclic graphs (Sec. 3.1)	25%
No group fine analysis (Sec. 3.2)	15%
No agreement analysis (Sec. 3.3)	52%
Agreement tracking only for `active`	37%

The results show that the agreement analysis is the key ingredient for the WCET analysis precision. We can also see that this analysis needs to be done not only for the active threads but also the pending threads. Eventually, we can spot that a precise group analysis is less crucial but should not be neglected.

5 State of the art

The problem of deriving a safe WCET bound for GPU kernels has so far been the topic of only a limited number of publications.

In [4], the authors tackle the problem by providing an ILP formulation that captures how a work-conserving warp scheduler could handle the workload corresponding to a kernel. The focus is put on how the scheduler hides the long latency instructions (e.g. memory accesses). However, in this preliminary work, the simplifying assumptions are very strong. In particular, the considered kernels are single path, which greatly simplifies the analysis and completely puts aside the problematic of thread divergence. The method proposed in [10] is also based on the modelling of the warp scheduler policy using an ILP system, focusing on the Greedy Then Round Robin scheduling policy. The authors propose analyses of the kernel code to handle memory access coalescing and roughly account for thread divergence. However, nothing is said about how thread divergence is detected, so we can assume that additional analyses such as the one that we propose are required.

In [5], a hybrid analysis method is used: a CFG is built to represent the execution paths of the kernel, and the execution duration of the basic blocks of the CFG are obtained using measurements. The authors propose an algorithm that extends the CFG of a single thread to a CFG that over-estimates the control transitions at warp-level, by adding extra edges that model thread divergence. This method is based solely on the topological properties of the thread CFG (i.e. divergence in the graph and dominance/post-dominance properties), and does not take into account the actual SIMT semantics, nor the fact that thread reconvergence only happens if and when SYNC/BRK instructions are inserted by the compiler. In our experience, the compiler does not always insert reconvergence instructions at the first post-dominant point in the CFG after a separation of paths in the graph, so the assumptions made in [5] may sometimes be too optimistic and lead to underestimations of the WCET. Moreover, since the authors do not follow closely the SIMT semantics, their method may add extra edges that are not added with our method a) when we detect an agreement between the threads of a warp and b) because when a conditional branch occurs, their algorithm does not know which branch is taken first.

An ad-hoc WCET analysis method for GPU kernels has been proposed in [9]. The algorithm builds on Single Static Assignment (SSA)-like analysis methods and on symbolical execution to statically detect the points in the CFG of a thread where agreement between all threads of a warp is statically guaranteed. This is, to the best of our knowledge, the only method (before ours) that tries to determine agreement on a condition when a conditional branch occurs. In comparison, our method determines agreement points by introducing the SIMT stack mechanism in our model, which provides more precise results: we determine agreement between active threads at each point, while the method based on SSA-like properties only allows to reason about all the threads in the warp. Moreover, our method aims at building a warp-level CFG that can then be used in a standard WCET analysis pipeline, so it can benefit from classical analyses on CFGs (e.g. cache analysis) and from the IPET method, while the method of [9] is standalone.

6 Conclusion

We presented an abstract interpretation technique to automatically build a warp-level CFG for GPU kernels. Our method strictly follows the SIMT semantics as implemented in the Nvidia Pascal GPUs. In particular, it handles the possibility of thread divergence and its impact on the warp-level CFG. Part of our analysis focuses on the representation of

agreement between threads of the same warp on a given value (e.g. a register value) to filter out divergence when we can statically prove that all threads agree on the branch to take. We performed an evaluation on the Rodinia benchmark, and highlighted the importance of the agreement analysis.

In the future we will improve our analysis by supporting the calls to kernels from other kernels, and by adapting classical analyses (e.g. loop bound analysis) to our framework. We will also work on our understanding of the microarchitecture of GPU targets in order to derive precise durations for the blocks of the CFG.

References

1 Sebastian Altmeyer and Claire Maiza. Cache-related preemption delay via useful cache blocks: Survey and redefinition. *J. Syst. Archit.*, 57(7):707–719, 2011. doi:10.1016/j.sysarc.2010.08.006.

2 T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith. Gpu scheduling on the nvidia tx2: Hidden details revealed. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, 2017.

3 Gogul Balakrishnan and Thomas W. Reps. WYSINWYX: what you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32(6):23:1–23:84, 2010. doi:10.1145/1749608.1749612.

4 Kostiantyn Berezovskyi, Konstantinos Bletsas, and Björn Andersson. Makespan Computation for GPU Threads Running on a Single Streaming Multiprocessor. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 277–286, July 2012. ISSN: 2377-5998. doi:10.1109/ECRTS.2012.16.

5 Adam Betts and Alastair Donaldson. Estimating the WCET of GPU-Accelerated Applications Using Hybrid Analysis. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 193–202, July 2013. ISSN: 2377-5998. doi:10.1109/ECRTS.2013.29.

6 François Bourdoncle. Efficient chaotic iteration strategies with widenings. In Dines Bjørner, Manfred Broy, and Igor V. Pottosin, editors, *Formal Methods in Programming and Their Applications, International Conference, Akademgorodok, Novosibirsk, Russia, June 28 - July 2, 1993, Proceedings*, volume 735 of *Lecture Notes in Computer Science*, pages 128–141. Springer, 1993. doi:10.1007/BFb0039704.

7 Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, October 2009. doi:10.1109/IISWC.2009.5306797.

8 Robert I. Davis, Sebastian Altmeyer, Leandro Soares Indrusiak, Claire Maiza, Vincent Nélis, and Jan Reineke. An extensible framework for multicore response time analysis. *Real Time Syst.*, 54(3):607–661, 2018. doi:10.1007/s11241-017-9285-4.

9 Vesa Hirvisalo. On Static Timing Analysis of GPU Kernels. In Heiko Falk, editor, *14th International Workshop on Worst-Case Execution Time Analysis*, volume 39 of *OpenAccess Series in Informatics (OASIcs)*, pages 43–52, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISSN: 2190-6807. doi:10.4230/OASIcs.WCET.2014.43.

10 Yijie Huangfu and Wei Zhang. Static WCET Analysis of GPUs with Predictable Warp Scheduling. In *2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC)*, pages 101–108, May 2017. ISSN: 2375-5261. doi:10.1109/ISORC.2017.24.

11 Rémi Meunier, Thomas Carle, and Thierry Monteil. Correctness and Efficiency Criteria for the Multi-Phase Task Model. In Martina Maggio, editor, *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*, volume 231 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:21, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECRTS.2022.9.

12 I. S. Olmedo, N. Capodieci, J. L. Martinez, A. Marongiu, and M. Bertogna. Dissecting the cuda scheduling hierarchy: a performance and predictability perspective. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020.

A Appendix

A.1 If-then-else example

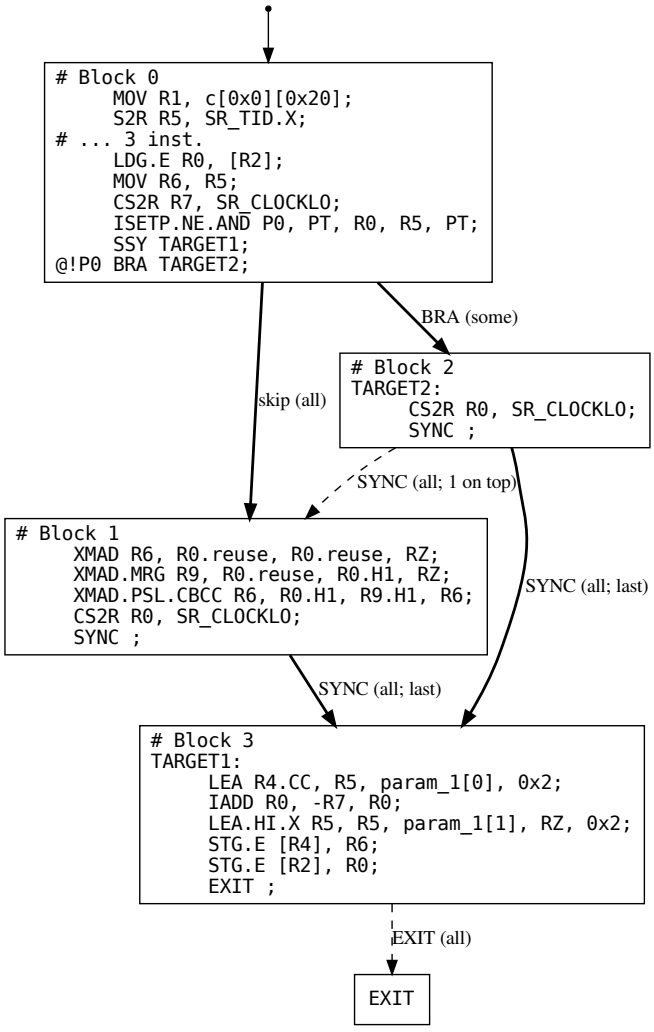

Figure 2 Control flow graph of a program containing a non-trivial if-then-else. The dashed control edge is not part of the source control flow.

A.2 Regular loops example

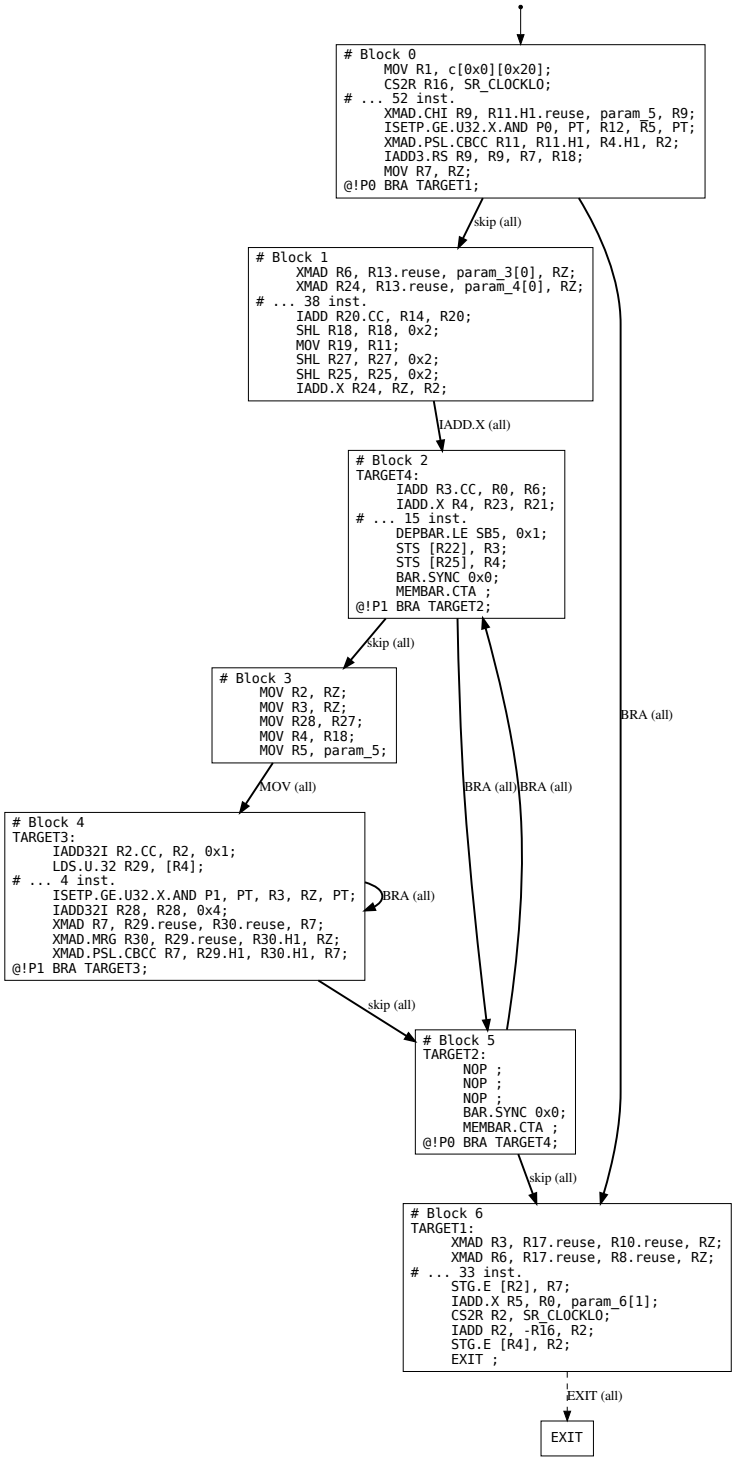

Figure 3 Control flow of a tiled matrix multiplication. Without agreement analysis, the exit instruction in block 6 would have an edge to every potential divergence.