On Static Timing Analysis of GPU Kernels

Vesa Hirvisalo

Aalto University
Espoo, Finland
vesa.hirvisalo@aalto.fi

Abstract

We study static timing analysis of programs running on GPU accelerators. Such programs follow a data parallel programming model that allows massive parallelism on manycore processors. Data parallel programming and GPUs as accelerators have received wide use during the recent years.

The timing analysis of programs running on single core machines is well known and applied also in practice. However for multicore and manycore machines, timing analysis presents a significant but yet not properly solved problem.

In this paper, we present static timing analysis of GPU kernels based on a method that we call abstract CTA simulation. Cooperative Thread Arrays (CTA) are the basic execution structure that GPU devices use in their operation that proceeds in thread groups called warps. Abstract CTA simulation is based on static analysis of thread divergence in warps and their abstract scheduling.

1998 ACM Subject Classification F.3.2 Semantics of Programming Languages

Keywords and phrases Parallelism, WCET

Digital Object Identifier 10.4230/OASIcs.WCET.2014.43

1 Introduction

In this paper, we study the timing of data parallel kernels executed on SIMT machines. The SIMT execution model (Single Instruction Multiple Threads) is typical for the abundant GPUs (Graphics Processing Units) that have become the main platform for massively parallel computing. In addition to their original purpose, graphics, GPUs are used as general purpose computing devices for applications covering a wide range from super computing to ordinary desktop computing.

GPUs are used also in some real-time applications, but their wider application for such purposes is limited by the lack of solid timing analysis methods. Especially considering safety critical applications, methods giving run-time guarantees are a must. Many embedded systems found in cars, airplanes, medical instruments, etc. are safety critical. As traditional single-core processors have ceased to scale effectively in their performance, multicore and manycore processors are needed in order to cope with the increasing computational demands of novel applications.

In this paper, we consider the timing of kernels executed on SIMT machines and focus on worst-case execution time (WCET) analysis. We make the following contributions:

- we present an abstract model of SIMT execution that is suitable for WCET analysis,
- a static WCET analysis method for SIMT executed data parallel kernels, and
- a WCET analysis of an example kernel that captures typical GPU computation.

We have structured our presentation in the following way. We start with a background section that describes GPU computation. In our background description, we use a short example kernel that we use also as a running example through the paper. We continue by describing our method in a section that is divided into subsections according to the main phases of our method: static divergence analysis, abstract warp construction, and the abstract CTA simulation itself. After that, we present an abstract GPU machine that we call mini-SIMT and present how our example program can be executed on the machine. This is followed by our WCET analysis applied to the program. We end our paper with a short review of related work and our conclusions.

2 Background

Graphics Processing Units (GPU) can be used as accelerators for general purpose computing in a setting that is is called GPGPU computing. GPUs are not stand-alone devices. They need a host computer to operate.

The basic operation in GPGPU computing consists of normal thread execution on the CPU of the host computer. Computationally heavy operations are *launched* to the GPU to be executed. Launches follow the abstract architecture of GPU devices, which forms the basis of GPGPU programming as exemplified by CUDA and OpenCL. The programs in such languages include code for both the host CPU and the accelerator GPU [7, 6].

Abstractly, GPUs consist of several processors with several cores each. Each of the processors has a fast local memory that its cores share. The processors communicate with each other by using a slow and large global memory.

GPU launches consist of a massive number of threads that are explicitly partitioned into blocks by the programmer. The processors on a GPU execute the blocks in any order and without synchronization until no blocks remain unexecuted. We will ignore the block structure of GPGPU programs in the rest of this paper. A broader view can be found, e.g., in [1].

2.1 Kernels

The GPU side code of GPGPU programs consist of *kernels*. A kernel defines the computation done by a single thread during a launch. The kernels belonging to the same block operate in a shared memory. For simplicity, we assume a launch to consist of a single block in the following and the GPU to consist of a single processor.

`TriangleSum` (Listing 1) is an example of a kernel. It is somewhat artificial, but it contains in a compact form many of the properties that are typical for GPGPU programming. The kernel is written in a language resembling the C language variant used in OpenCL.

To use the kernel, it must be launched. Consider, for example, a 16x16 matrix `m` to be processed by the kernel into a triangle column sum vector `v` of length 16. On the CPU side we would execute the code in Listing 2 to make a launch.

The launch causes 16 threads to be started on the GPU accelerator as indicated by `size` between the angle brackets that mark the call as an accelerator call. The 16x16 matrix `m` is in the row major layout.

Inside the kernel, we have a special variable `Tid` that is defined by the system as the number of the thread executing (e.g., if thread 0 evaluates `Tid`, it gets the value 0). As `i` is initialized to value `Tid` and incremented by `c` (i.e., 16), the reference `m[i]` will cause each thread to access the element (Tid, j) of the matrix, where j is the iteration number.

As can be seen from the code, different threads execute different number of iterations of the for loop. Only the upper right triangle of the matrix gets accessed. Further, only every second row will be summed because of the condition `d % 2` and an adjustment will be added to some elements because of the condition `d % (Tid + 1) == 0`.

Listing 1 An example kernel.

```
__kernel TriangleSum(float* m, float* v, int c) {
  int d = 0;                 /* each thread has its own variables */
  float s = 0;                    /* s is the sum to be collected */
  int L = (Tid + 1) * c;
  for (int i = Tid; i < L; i += c) {
    if ((d % (Tid + 1) == 0)
      s += 1;
    if (d % 2)
      s += m[i];
    __syncthreads();         /* assuming compiler support for this */
    d += 1;
  }
  v[d-1] = s;
}
```

Listing 2 A launch of the example kernel.

```
size = 16;
float m[size] = {0, .., 255};
float v[size];
TriangeSum<size>(m, v, size);
```

By default, the threads are free to proceed at their own pace. However, to synchronize the threads the programmer has added a barrier. All active threads will wait each other at `__syncthreads()` and proceed only after all of them have reached the barrier. After the loop, each thread writes the adjusted sum of one triangle column to the result vector `v`.

2.2 SIMT execution model

GPU devices use the Single Instruction Multiple Threads (SIMT) execution model. In the model, threads are grouped into warps of constant size. For example, with the warp size 8, our example launch would consist of two warps: one for threads 0...7 and the other for threads 8...15.

In the SIMT model, each warp is processed by executing the same instruction for all of its threads. In this respect, the SIMT model resembles typical vector computing (the SIMD model). For example, the addition `s += m[i]` is a single instruction but does eight operations in parallel.

However, all threads have their own program counter. Thus, the processing of threads can differ if there is branching. In the SIMT execution model, the different progress of threads is called *thread divergence*.

Simple if statements can be handled by predicated execution. For example, the value of the if statement condition `d % (Tid + 1) == 0` can be computed in parallel for all the threads in a warp and a corresponding execution mask can be formed. The following addition statement has effect only for the threads marked active by the mask.

Nested branching, such as nested if statements, is more complicated to handle. For example, the code in Listing 3 has two nested divergences. Because of `p1`, some threads are executing `s3`. The others are split by `p2` to execute the different subbranches.

To handle such structures, stack-based reconvergence mechanisms are typically used by GPU hardware. Let the condition p1 to evaluate to the vector (1, 1, 0, 0, 0, 0, 1, 1). Thus,

Listing 3 Nested branching

```
if (p1)          /* A */
  if (p2) s1     /* B */
  else s2
else s3          /* C */
...              /* D */
```

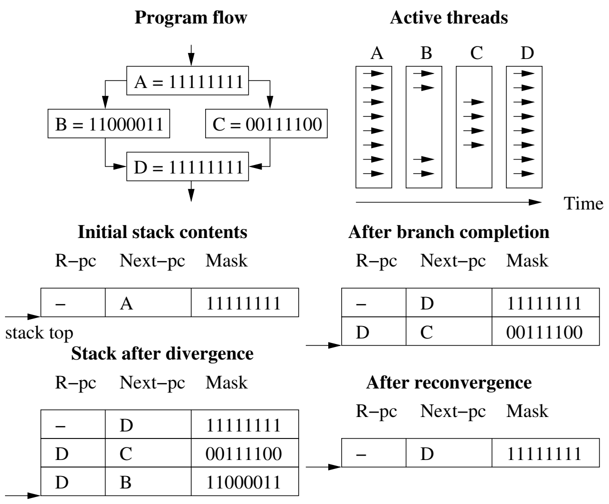


Figure 1 Thread divergence handling.

threads 3...6 should evaluate the else branch labelled `C`. The GPU hardware pushes the mask 00111100 onto a reconvergence stack together with the PC value `C` and reconvergence PC value `D`. The reconvergence PC value is needed to mark the instruction at which the divergent threads meet. The processing sequence is illustrated in Figure 1.

2.3 Avoiding memory latencies

Execution can be stalled by an instruction reading memory. GPU hardware hides such latencies by allowing multiple warps to be run concurrently. All the warps from a single block are collected into a Cooperative Thread Array (CTA). The scheduling hardware of the GPU keeps track of ready warps of a CTA, i.e., warps that are not stalled because of memory accesses (or other reasons). Typically round-robin scheduling policy is used and the scheduler is able to keep in pace with instruction issuing.

Considering our example program, the benefit of round-robin scheduling can be easily seen. By default, the warps get the same amount of execution cycles from the processor. The warps executing `TriangleSum` will progress almost with the same speed through the code and will also reach the synchronization barrier `__syncthreads()` almost simultaneously at each iteration. Thus, the barrier wait time is not very large.

We define *occupancy* as the number of ready warps. The memory latency hiding is dependent on the occupancy of the processor. As the warps may be partially filled, occupancy may differ from thread utilization.[1]

[1] Occupancy is usually defined as the percentage $actual\_warps/max\_warps$ (but we will use the warp count) and current hardware can support 64 warps of 32 threads (but we will use modest numbers).

3 WCET estimation method

In describing our WCET estimation method, we concentrate on features related to the SIMT execution model and omit the other features (e.g., pipelines and caches). We define the total time spent in execution as

$$T_{exec} = T_{instr} + T_{stall}$$

where T_{instr} is the number of cycles spent executing instructions and T_{stall} is the number of cycles spent in instruction execution stalls because of local memory waits.

Even assuming a simplistic hardware model, T_{instr} cannot be directly counted from the code, because the SIMT execution model allows divergence. Considering an if-else structure, the execution time is

$$T_{if\_else} = \begin{cases} T_{true\_branch} & \text{if all threads converge to true} \\ T_{false\_branch} & \text{if all threads converge to false} \\ T_{false\_branch} + T_{true\_branch} & \text{if threads diverge} \end{cases}$$

Considering loops, the execution time of a warp is the time of the longest thread in the warp. Thus, for control structures, the execution time is dependent on divergence. To estimate the WCET statically, we need static divergence analysis.

Warp scheduling hides the memory latencies. In the worst case, all warps execute a memory read on consecutive cycles and the stall is

$$T_{stall} = \max(0, T_{memory} - N_{warps})$$

where T_{memory} is the memory access latency and N_{warps} is the warp occupancy. Note that T_{stall} is directly added to the total execution time, not to individual warps.

To get tight timing estimates for SIMT executed programs, we must be able to statically estimate occupancy. We base the estimation on a method that we call abstract CTA simulation. Abstract CTA simulation needs abstract warps to be constructed.

In the following, we will first describe divergence analysis, then abstract warp construction, and finally abstract CTA simulation.

3.1 Static divergence analysis

We base our static divergence analysis on GSA (Gated Single Assignment). It augments programs with value chaining information and resembles SSA (Static Single Assignment). Instead of the ϕ-function of SSA it uses three special functions to build the chains:

- γ function is a join for branches. $\gamma(p, v_1, v_2)$ is v_1 if the p is true (or else v_2).
- μ function is a join for loop headers. $\mu(v_1, v_2)$ is v_1 for the 1^{st} iteration and v_2 otherwise.
- η is the loop exit function $\eta(p, v)$. It chains a loop dependent value v to loop predicate p.

GSA allows control dependencies to be transformed into data dependencies whose chains we can follow. We say that a definition of a variable is *divergent* if the value is dependent on the thread. If there are no divergent definitions for a branch predicate, we know the branch to be non-divergent. The details of the method can be found in [3].

3.2 Abstract warp construction

For static WCET estimation, we need timing information of the warps to be executed. We do this by constructing abstract warps. Abstract warps resemble the traditional control flow graphs. An *abstract warp* $A = (V, E)$ is directed graph. The nodes V have three node types:

- Time nodes describe code regions with two values. T_{instr} is the upper bound of the instruction execution time consumed. T_{shift} is the upper bound of the variation of the instruction execution time caused by thread divergence.
- Memory access nodes that mark places where memory access stalls may happen.
- Barrier nodes that mark places where barrier synchronization must happen.[2]

An abstract warp is constructed from the code in a recursive bottom-up way by applying:

```
procedure construct(program element P)
  case P
    is compound statement $S_1, S_2$: R = Join(construct($S_1$), construct($S_2$))
    is if statement for $S_1, S_2$: R = LUB( construct($S_1$), construct($S_2$))
    is loop statement with body $S$: R = Cons_loop_edge(construct($S$))
    is memory read statement $S$: R = Cons_flow_edge(Time($S$), Memory())
    is memory barrier statement $S$: R = Cons_flow_edge(Time($S$), Barrier())
    is some other $S$: R = Time($S$)
```

where R is the return value. The constructor Join merges S_1 and S_2 by summing the T_{instr} and T_{shift} values, if they are statically resolvable. The constructor LUB merges S_1 and S_2 by selecting the worst-case T_{instr} and the T_{shift} values, if they are statically resolvable. The constructors Time, Memory, and Barrier construct simple nodes.

3.3 Abstract CTA simulation

We use abstract CTA simulation to get execution time estimates for the kernels whose structure we can statically resolve. Instead of exhaustively executing all warps with thread masks, convergence mechanisms, and all possible scheduling interleaving choices, abstract CTA simulation considers a single abstract warp. It uses static estimation to understand the effects of multiple warps, thread divergence, memory latencies and scheduling choices.

Our abstract CTA simulation assumes the hardware to use round-robin scheduling for warps. This means that without divergence between the warps, the warps will be executed as *convoys*. A convoy is the execution of the same instruction by all warps in a single round-robin cycle. We define divergence from this scheduling as *convoying shift* T_{SHIFT}, which is the program counter distance among threads within a single round-robin cycle.

The WCET calculation is based on cumulative sum of instruction execution and memory stall times during the abstract CTA simulation. In the simulation, we use bounds for occupancy: N_{low} is the lower bound and N_{high} is the upper bound.

```
procedure simulate(abstract warp A, warp count WC)
  $T_{WCET} = 0$
  $N_{low} = N_{high} = WC$
  proceed through A = (V, E) until termination
    let $v \in V$ be the current node
    case v
      is time node:
        $T_{WCET} \mathrel{+}= N_{high} * T_{instr}$
        $T_{SHIFT} \mathrel{+}= T_{shift}$
```

[2] Note that barriers can cause deadlocks in actual programs. We assume programs to be deadlock-free.

```
    is barrier node:
      reset $T_{SHIFT}$ and update $N_{low}$ and $N_{high}$
    is memory access node:
      add the access to $LOG$
      increment $T_{SHIFT}$ according to $N_{low}$
      update $N_{low}$ and $N_{high}$
  flush accesses from the $LOG$
```

The simulation of memory access interleavings is done by keeping a LOG, whose length is limited by T_{SHIFT} and the branching encountered during the simulation. When we flush an access from LOG, we increment T_{WCET} by T_{stall}. In computing T_{stall}, we pessimistically consider the interleavings that can happen within the LOG. Thus, instead of considering whole programs (see, e.g., [5]), we consider interleavings within a small window.

In actual hardware, warps can execute a kernel without synchronization. Our interleaving mechanisms can handle only modest convoying shifts. We can accurately simulate some loop types. Most importantly, these include the loops for which the loop branch is non-divergent.

4 The mini-SIMT machine

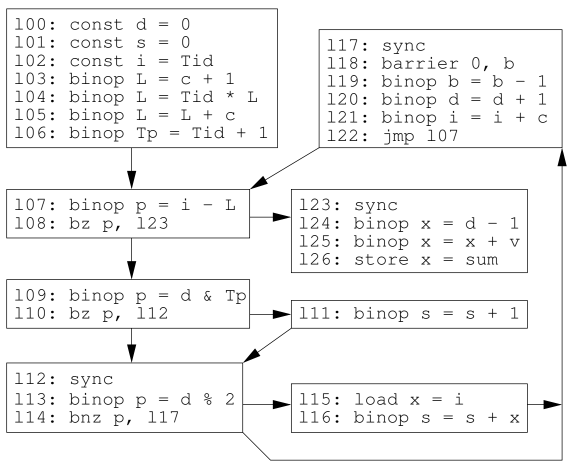
We will use a simple machine language (inspired by [3]):

```
Labels (L)                       ::=   l ∈ N
Variables (V)                    ::=   Tid ∪ {v1,v2,...}
Instructions                     ::=
- (jump if zero/not zero)         |    bz/bnz v,l
- (unconditional jump)            |    jump l
- (store into shared memory)      |    store v_x = v
- (load from shared memory)       |    load v = v_x
- (arithmetic operation)          |    binop v_1 = v_2 op v_3
- (immediate copy)                |    const v = n
- (re-convergence)                |    sync
- (synchronization barrier)       |    barrier v_1, v_2
```

A mini-SIMT program executes kernels with SIMT execution model with multiple cores and uses round-robin scheduling to schedule ready warps. For each warp, the machine keeps a synchronization stack holding frames $(l_{id}, \Theta_{done}, l_{next}, \Theta_{todo})$, where l_{id} is the conditional branch that caused the divergence, Θ_{done} is the set of cores that have reached the synchronization point, l_{next} is the instruction where the set of cores Θ_{todo} will resume execution.

The machine pushes frames at branches (`bz` or `bnz`) and pops them at reconvergence points, but uses a separate reconvergence instruction `sync` to mark the reconvergence points (instead of pushing reconvergence PC onto a stack). Only `load` can cause memory stalls. The `barrier` instruction unschedules the warps that have reached the barrier until v_2 threads are the same barrier (identified by v_1).

The mini-SIMT machine code for our example kernel and its control flow graph is given in Figure 2, whose right side lists excerpts of the kernel executed with 4 threads in a single warp, instruction execution time of 1 cycle, and memory latency of 10 cycles



```
100: const d = 0
101: const s = 0
102: const i = Tid
103: binop L = c + 1
104: binop L = Tid * L
105: binop L = L + c
106: binop Tp = Tid + 1
```

```
117: sync
118: barrier 0, b
119: binop b = b - 1
120: binop d = d + 1
121: binop i = i + c
122: jmp 107
```

```
107: binop p = i - L
108: bz p, 123
```

```
123: sync
124: binop x = d - 1
125: binop x = x + v
126: store x = sum
```

```
109: binop p = d & Tp
110: bz p, 112
```

```
111: binop s = s + 1
```

```
112: sync
113: binop p = d % 2
114: bnz p, 117
```

```
115: load x = i
116: binop s = s + x
```

Cycle	PC: instruction	mask
00	100: const	1111
	...	
25	115: load	1111
26	115:	stall
27	115:	stall
	...	
33	108: bz	1111
34	123: sync	1000
35	109: binop	0111
	...	
86	108: bz	0001
87	123: sync	0001
88	124: binop	1111
89	125: binop	1111
90	126: store	1111

Figure 2 The control flow graph of our translated example program (left) and parts its execution with 4 threads (right), 1 = active thread, 0 = passive thread. Note the pushes and pops by `sync`.

5 An example of analysis

To clarify our method, we consider the program in Listing 1 to be executed on a mini-SIMT machine with 16 threads in 4 warps. In the following, we will first do divergence analysis for the kernel, then we will construct an abstract warp that describes the kernel, and finally, do abstract CTA simulation for it.

5.1 Static divergence analysis

We use GSA in the divergence analysis. Consider the branch at `108`. GSA places a μ function at `107` that chains the branch predicate to the definition (`102: const i = Tid`). As `Tid` is a divergent value, the loop is divergent. Similarly, the branch at `110` is divergent because its predicate is chained to the definition (`106: const i = Tid + 1`).

The branch at `114` is not divergent. GSA places a μ function at `107` that chains the branch predicate to the definitions (`100: const d = 0`) and (`120: const d = d + 1`). Both definitions are non-divergent. Thus, the branch itself and also the memory access controlled by the branch are non-divergent.

5.2 Abstract warp construction

Our abstract warp construction basically follows the CFG given in Figure 2. The number of instructions in the basic blocks give us the related T_{instr} values, because of the execution time of 1 cycle for all instructions. The construction algorithm splits the block of `115` in two and places a memory access node in between, because the block contains a memory read. Similar splitting happens for the block of `118`, into which a barrier node is added.

Some nodes can be merged together. Especially, the path (`109`, `110`, `111`, `112`, `113`, `114`, `115`) is interesting. It is a worst-case path with divergence at `111`. The corresponding time node gets the values $T_{instr} = 7$ and $T_{shift} = 1$. The resulting abstract warp is in Figure 3.

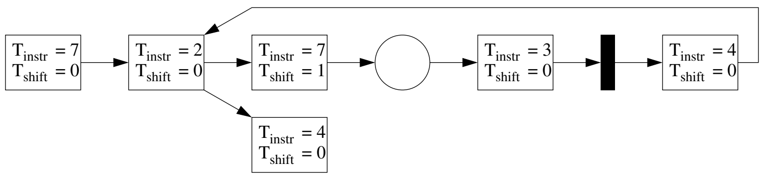


Figure 3 Abstract warp of the example kernel, where boxes are time nodes, the circle is a memory access node, and the bar is a barrier node.

5.3 Abstract CTA simulation

The abstract CTA simulation begins from the leftmost node in Figure 3. As the warp width is 4, we have 4 warps. Thus, the first node adds $7 * 4 = 28$ cycles to T_{WCET} and sets $T_{shift} = 0$. The simulation processes similarly for the next time node.

After that we encounter a divergent branch. Because of the static loop branching predicate we are able to resolve that the iteration counts for the warps are (4, 8, 12, 16). Thus, we proceed with $N_{low} = N_{high} = 4$.

$T_{SHIFT} = 1$ when we encounter the memory access node. The memory access will set $T_{SHIFT} = 3$. Such increasing shift would cause problems later in the simulation, but the barrier node will reset T_{SHIFT} to 0 and flush the read from LOG by increasing T_{WCET} by $10 - N_{low} = 6$ stall cycles.

After four iterations, one warp will diverge from the loop and the simulation continues in the loop with $N_{low} = N_{high} = 3$. After exiting the loop, the simulation continues with $N_{low} = 1$ and $N_{high} = 4$ for the remaining block yielding a final estimate $T_{WCET} = 804$. By using a cycle accurate simulator we obtain 688 as the true execution time.

6 Related work

The timing analysis of programs running on single core machines is rather well known. A survey of the WCET methods applicable for such purposes can be found in [8]. The common methods can be roughly divided into measurement-based methods and methods based on static program analysis. The static program analysis methods typically divide the WCET analysis problem into three sub-problems: flow analysis, processor behavior analysis, and WCET estimate calculation. Some methods have applied model checking, e.g., [4].

Recently, there has been a rising interest on WCET analysis targeting multicore platforms. For example, Gustavsson et. al. [5] present a timing analysis of multithreaded programs on a multicore computer. Their approach applies abstract execution to rather unrestricted programming model. Chattopadhyay et. al. [2] present a unified framework for WCET analysis. The framework is to tackle the problems that have arisen, when the classical approaches have been applied to multicore machines. There has been some work addressing WCET analysis of GPU computing, such as [1].

7 Conclusion

In this paper, we present static timing analysis of GPU programs based on a method that we call abstract CTA simulation. Abstract CTA simulation is based on static analysis of thread divergence in warps and their abstract scheduling.

Our method has obvious limitations. The static divergence analysis can give false positives that lead to over-estimation of execution time. Further, our handling of loops is simplistic.

For divergent loops it can give pessimistic timing, especially when there is complex branching in a kernel.

However, according to our own experience in GPGPU programming, typical kernels are simple in their structure. Many kernels do computation in a map-reduce style, where the mapping phase is essentially non-divergent and the reduction phase is divergent. Often, the actual occupancy is high for the mapping phase and low for the reduction phase. Our approach fits the analysis of such kernels.

Despite its short comings, our methods is very scalable. It can be used to analyze the WCET of very large numbers of parallel threads. This is caused by the fact that the abstract CTA simulation captures efficiently the timing of parallel threads. Abstract CTA simulation spends time in resolving iterations, but typical GPU kernels are short as they rely on massive parallelism.

Currently, our method lacks formal proof of correctness. Further, its applicability is limited by the fact that it has not been integrated with a traditional WCET estimation of the CPU side. Thus only kernels can be analyzed instead of full programs. We see these aspects as the most important topics for further research.

References

1 A. Betts and A. F. Donaldson. Estimating the WCET of GPU-Accelerated Applications Using Hybrid Analysis. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECTRS)*, pages 193–202, 2012.

2 S. Chattopadhyay, L. K. Chong, A. Roychoudhury, T. Kelter, P. Marwedel, and H. Falk. A Unified WCET Analysis Framework for Multi-core Platforms. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s), April 2014.

3 B. Coutinho, D. Sampaio, F. M. Q. Pereira, and W. Jr. Meira. Divergence Analysis and Optimizations. In *Proceedings of the International Conference on Parallel Architectures and Compilation (PACT)*, pages 320–329, 2011.

4 A. E. Dalsgaard, M. C. Olesen, M. Toft, R. R. Hansen, and K. G. Larsen. METAMOC: Modular Execution Time Analysis using Model Checking. In *Proceedings of the International Workshop on Worst-Case Execution Time Analysis (WCET)*, pages 114–124, 2010.

5 A. Gustavsson, J. Gustafsson, and B. Lisper. Timing Analysis of Parallel Software Using Abstract Execution. In *Proceedings of International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 59–77, 2014.

6 Khronos. OpenCL documentation. `http://www.khronos.org/opencl/`.

7 NVIDIA. CUDA documentation. `http://nvidia.com/`.

8 R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, April 2008.