Estimating the WCET of GPU-Accelerated Applications using Hybrid Analysis

Adam Betts and Alastair Donaldson
Department of Computing
Imperial College London
London SW7 2AZ
Email: {adam.betts,alastair.donaldson}@imperial.ac.uk

Abstract*—The massive parallelism offered by Graphics Processing Units (GPUs) is now routinely exploited to accelerate computationally intensive tasks in a wide variety of application domains. Efficient GPU programming in languages such as CUDA and OpenCL requires careful application of hand optimisations to exploit parallelism and locality while minimising synchronisation. The effectiveness of such optimisations can be highly dependent on workload and the structure of input data, making it difficult to assess performance in general by testing alone. To address this, we study the problem of estimating the Worst-Case Execution Time (WCET) of GPU-accelerated applications. We propose the use of *hybrid* WCET analysis whereby execution times of small program segments are deduced from traces of execution and a calculation backend derived from the Control Flow Graph (CFG) produces a WCET estimate. Standard techniques which construct a CFG from a binary cannot be applied directly to GPU code because they miss implicit execution paths that arise due the way branches are implemented in hardware — we present a solution using standard compiler analysis. We further describe how to extend the basic hybrid WCET analysis of sequential code so that concurrent timing effects in the GPU execution model are incorporated. We have implemented our analysis as a tool built on top of the GPGPU-sim open source simulator. We evaluate our tool using a set of benchmarks drawn from the CUDA SDK: results show that effective modelling of concurrency is key to reducing pessimism in the WCET calculation.

I. Introduction

Graphics Processing Units (GPUs) are highly *parallel* architectures, using features of Single-Instruction, Multiple-Data (SIMD) execution, Multiple-Instruction, Multiple-Data (MIMD) execution, and Instruction-Level Parallelism (ILP) to deliver high performance [1]. Due to their raw compute power — and with the advent of CUDA [2] and OpenCL [3] helping to ease the programming burden — GPUs are now routinely used as accelerators in fields such as bioinformatics, computer vision, weather forecasting, and medical imaging.

As in the case of CPU programming, ensuring that a GPU application efficiently utilises computational resources is a cardinal goal. Most often this involves analysing *average-case* performance and optimising accordingly, but outlier execution times, such as the **Worst-Case Execution Time** (WCET), also prove fruitful. For instance, the emerging practice is to write parallel applications at higher levels of abstraction using Domain-Specific Languages (DSLs) and to generate the GPU implementation with a DSL compiler [4]. These DSL compilers aim for performance portability across a diverse range of GPUs, typically achieved through *auto-tuning*; that is, by generating several code variants and, through analytical models or empirical evidence, selecting the "best" for a specific GPU implementation. The auto-tuning stage can be greatly aided by WCET estimation because it helps to diagnose potential performance bottlenecks and serves to break ties when average execution times among several alternatives are (almost) equivalent. Computing WCET estimates is also crucial to the construction of *real-time systems* since they are assumed as input to the vast majority of scheduling algorithms [5]. Indeed, subsequent verification that the schedule completes within a specific period is only valid when the WCET estimates bound the *actual* (non-computable) WCETs. Thus, integration of GPUs into real-time systems requires suitable support from WCET techniques and tools.

This paper presents the first work on applying WCET analysis to the analysis of GPU-accelerated software, a previously uninvestigated area [6]. We focus on applications running on the NVIDIA [7] family of GPUs, though our novel techniques are applicable to any GPU utilising a similar execution model. Despite the fact that research in WCET analysis for single-core, uniprocessor architectures has reached a certain level of maturity [8], these methods cannot be directly applied to GPU software due to three principal problems: the limitations of static analysis, the complexity of the *lock-step* execution model employed by mainstream GPUs, and the requirement to accurately reason about bulk-synchronous parallel execution. We present novel solutions to each of these problems, resulting in a hybrid WCET analysis which we have implemented as a practical tool and evaluate across a range of benchmarks.

We begin by discussing the above problems in a little more detail and summarising the results of our evaluation.

Problem One: Static Analysis

WCET analysis is presently dominated by *static* analyses that require an *accurate* model of the hardware to obtain execution times of program segments; recently, Lisper [6] has argued that this sort of analysis is pliable to GPUs. Yet GPU manufacturers rarely reveal specific implementation details, which are vital to any model, in order to maintain their competitive edge. For instance, the associativity and replacement policies of the cache, the pipeline depth, and how exactly threads are scheduled on NVIDIA GPUs all remain undisclosed. For these reasons, our belief is that soundly modelling the hardware and its associated run-time support is infeasible. Consequently, we argue that GPUs are only

This work was supported by the EU FP7 project CARP (project number 287767)

applicable to firm, soft, or probabilistic [9] real-time systems where occasional deadline misses can be tolerated.

Our solution is to use *hybrid* techniques [10], [11] that can, in principle, avoid hardware modelling. In hybrid analysis, the execution times of small program segments are obtained from the *dynamic* profile of the program while *static* analysis techniques serve to stitch these execution times together. In particular, this paper builds upon the **Instrumentation Point Graph** (IPG) [10], [12] approach, pertinent details of which are reviewed in Section II.

Problem Two: Branch Divergence

The majority of WCET analyses adopt the **Control Flow Graph** (CFG), e.g. to obtain flow analysis data [13], to compute a WCET estimate [14], or to construct the IPG [10]. The salient feature of the CFG is that it is an over-approximation of the set of possible paths through the program, and any subsequent analysis on the CFG therefore yields an overestimation of the actual behaviour.

However, standard techniques [15]–[17] to obtain the CFG do not capture all possible execution paths through applications running on NVIDIA GPUs because of the way threads are executed on NVIDIA hardware (see Section III). Threads are grouped together into units of 32, termed *warps*, such that all threads within a warp share a program counter and hence follow the same execution path. When a conditional branch is encountered, it is possible that different threads within a warp want to explore different paths (so-called *branch divergence*). The hardware handles this situation by executing one side of the branch with a subset of threads until reaching a point where control flow re-converges, then backtracking to the other side of the branch with the remaining threads enabled. Control can effectively jump from a basic block that appears textually to have no branch instructions, to another basic block that appears at compile time not to be a branch target.

We show how the post-dominator relation [17] and data-flow analysis [15] can be used to insert additional edges into the CFG to correctly model branch divergence. In particular, we demonstrate that these edges can only be added *after* loops have been identified in the CFG, otherwise problems associated with irreducibility [18] will curtail the WCET analysis. Full details of the solution appear in Section IV.

Problem Three: Parallelism and Concurrency

Existing WCET analyses (including the approach based on the IPG) assume that tasks run with a *single* thread of control. However, this assumption does not hold for massively parallel GPU applications. Thus, any WCET analysis, or indeed timing analysis in general, must account for concurrency and synchronization to output reasonable estimates.

Specifically within the context of IPG-based analysis, the GPU execution model adds two particular complications. First, instrumentation events generated by distinct warps appear in an interleaved fashion in a trace, and we cannot deduce at compile time how these interleavings will manifest because the scheduling policy is unknown. Our solution is to *slice* the set of time-stamped traces into sets of *warp-specific traces* and then feed these into the standard IPG analysis framework. However, this is generally not sufficient to derive an upper bound on the end-to-end execution time because it does not account for *concurrent* timing effects, in particular the delay experienced by warps before they commence execution. We propose two solutions: one computes the worst-case release jitter experienced before a warp is dispatched to computational resources by analysing the time-stamped traces, whereas the other constructs an analytical model of how warps arrive in so-called *waves* and obtains values for this model from the time-stamped traces. How we slice traces and handle concurrent timing effects are presented in Section V.

Evaluation

We have implemented our techniques on top of GPGPU-sim [19], an open-source cycle-accurate GPU simulator that supports recent generations of NVIDIA hardware. Using our tool, we evaluate multiple CUDA applications shipped with the CUDA SDK [2]: Section VI presents results of our experiments, and the paper concludes in Section VII.

II. Background to Hybrid WCET Analysis

We begin with an overview of how hybrid WCET analysis works for *sequential* programs.

A. Instrumentation

Hybrid analysis inserts **Instrumentation Points** (Ipoints) into a program, the purpose of which is to time stamp execution at particular program points. We assume that there are always Ipoints at the start and end of the program, which we denote by s and t, respectively.

Instrumentation can take one of several forms: either software probes [20], hardware probes [21], or virtual probes with the support of a cycle-accurate simulator. Merits and downfalls of these options are discussed extensively elsewhere [22]. Our extension of hybrid WCET analysis to GPU code uses a cycle-accurate simulator, namely GPGPU-sim [19], because it offers great flexibility as to where Ipoints are placed and allows traces to be buffered and stored easily (see Section VI).

B. Trace Parsing and the IPG

The instrumented program then undergoes rigorous testing (see [23]) in order to stress the execution times of Ipoint transitions. The output of the testing phase is a set of *traces*. A trace is a sequence $(i_1, t_1), (i_2, t_2), \ldots, (i_n, t_n)$ of tuples generated by a single execution of a program, where for each $1 \leq j \leq n$, i_j is an Ipoint identifier and t_j its time of execution, and where $i_1 = \mathsf{s}$ and $i_n = \mathsf{t}$.

Hybrid analysis processes these traces to extract, at a minimum, the observed WCET of Ipoint transitions. In addition, it may also procure upper bounds on the number of loop iterations as required in the WCET calculation, though gathering any piece of data from the traces potentially renders the WCET calculation unsafe; an alternative is to utilise static high-level analyses [13] for path-related information.

Traces are parsed using an automaton, the IPG, which is simply a graph where vertices are Ipoints and transitions are a contraction of Ipoint-free paths between all Ipoint pairs in the instrumented program. Formally, an Ipoint-free path is a sequence $u \rightarrow b_1 \rightarrow \ldots \rightarrow b_n \rightarrow v$ such that every b_i is a basic block and *not* an Ipoint.

Although it is possible to build the IPG on the fly during trace parsing, generally it must be constructed *statically* from the structure of the CFG and the Ipoint locations within the CFG. The reason is that the WCET calculation phase requires loops and their nesting hierarchy to be identified in the IPG, but standard algorithms [18] fail when the IPG is *irreducible* [15], [24], i.e. when loops have multiple entries. Irreducibility is much more prevalent in the IPG because Ipoints are not always placed inside CFG loop headers. The solution [12] assumes that the CFG is reducible, and then infers from each CFG loop which Ipoint transitions create cycles in the IPG. A reducible CFG is therefore pivotal to IPG-based analyses — Section IV expands on complications with this assumption because of the way the GPU implements branch divergence.

Example: Consider Fig. 1, which shows a CFG with basic blocks as square vertices and Ipoints as circular vertices. Because there is an Ipoint-free path $i_2 \rightarrow b_2 \rightarrow b_3 \rightarrow i_2$ in the CFG, and Ipoint i_2 is both the source and sink of this path, there is an IPG edge $i_2 \rightarrow i_2$. Moreover, this edge is identified as an IPG loop because Ipoint i_2 is a loop header in the CFG. The other edges in the IPG are derived similarly.

The figure additionally includes two traces: each trace starts at i_1, i.e. $i_1 =$ s, and ends at i_4, i.e. $i_4 =$ t. Parsing of these traces with the IPG uncovers the following: $WCET(i_1 \rightarrow i_2) = 10$, $WCET(i_2 \rightarrow i_2) = 7$, $WCET(i_2 \rightarrow i_4) = 7$, $WCET(i_1 \rightarrow i_3) = 3$, $WCET(i_3 \rightarrow i_4) = 5$, and that $i_2 \rightarrow i_2$ iterates at most twice. All of this information is subsequently fed into the WCET calculation.

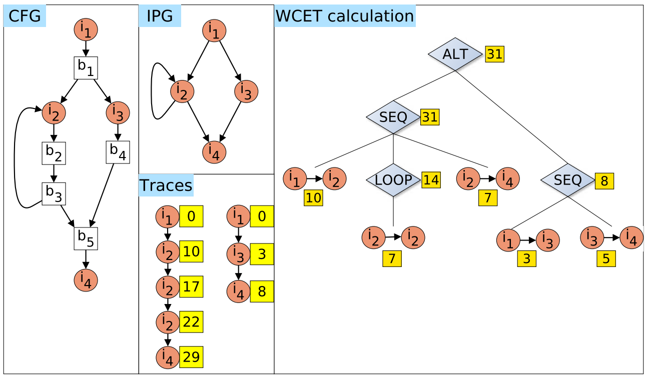


Fig. 1. Example of hybrid WCET analysis.

C. WCET Calculations on the IPG

Two approaches exist to compute a WCET estimate from an IPG: a tree-based approach [25] or through implicit path enumeration [10], [14]. Our toolset described in Section VI uses the latter because it is generally more accurate, but here we demonstrate the tree-based approach as it is more intuitive.

The IPG is first transformed into a tree representation that is similar to an abstract syntax tree: its internal vertices represent sequential, alternative, and iterative constructs, while leaves represent Ipoint transitions as these are the atomic units of computation. The tree is then traversed bottom up, combining the WCET values of every internal vertex's children using a specific rule: for a loop, multiply the body's WCET by the loop bound; for a sequence, sum the values; and for an alternative, take the maximum.

Example: Reconsider Fig. 1, which shows the tree representation of the IPG. Leaves in the tree have been annotated with their observed WCETs, as obtained from parsing the traces. The WCET of the loop vertex is $7 \cdot 2 = 14$ because the maximum execution count of $i_2 \rightarrow i_2$ is two. Finally, the WCET of the alternative (root) vertex is $max(31, 8) = 31$, which is the WCET estimate.

III. Background to CUDA and NVIDIA GPUs

This section gives a brief overview of the CUDA programming model (Section III-A) as well as the architecture and execution model of NVIDIA GPUs (Section III-B).

A. Programming Model

The CUDA programming model slices a code base into two disjoint parts: code intended for execution on the CPU (the **host**) is written in vanilla C/C++, whereas code intended for execution on the GPU (the **device**) is implemented in CUDA C/C++, which is a superset of a subset of C/C++. Its language extensions allow a programmer to label and launch a number of data-parallel functions called **kernels** which are intended for GPU execution. Every kernel is executed by hundreds or thousands of threads which, for reasons that become clear shortly, are partitioned by the programmer into a set of **thread-blocks** such that there is an equal number of threads per thread-block.

B. Architecture and Execution Model

The architecture of NVIDIA GPUs is evolving rapidly and giving sufficient coverage to each variation is beyond the scope of the paper. Thus the remainder of the paper concentrates on the Fermi generation of GPUs, although the principles of our techniques are applicable to all NVIDIA hardware.

A GPU consists of a number of **(streaming) multiprocessors**, each of which is composed of several lightweight **cores**. How many multiprocessors and cores actually reside on a GPU is implementation specific; for instance, the Fermi GF100 has 16 multiprocessors and 32 cores per multiprocessor, for a total of 512 cores overall.

During program execution, thread-blocks are assigned to specific multiprocessors. The number of thread-blocks that a multiprocessor can actively process depends on the upper bound imposed by the GPU implementation and on the resources (e.g. number of registers) consumed by a thread-block. The CUDA run time dynamically adjusts the number of thread-blocks assigned to a multiprocessor as its resources are occupied or relinquished. Moreover, since a kernel is typically launched with more thread-blocks than the multiprocessors can handle, the run time maintains a list of unserviced thread-blocks and dispatches them when multiprocessors become available. Thread-blocks therefore arrives in **waves** on each multiprocessor, which is an important property for our WCET analysis as explained in Section V.

Individual threads within a thread-block execute on a specific core. However, on an NVIDIA GPU, threads are not the atomic unit of scheduling — rather, it is a sub-group of the thread-block called a **warp**. The number of threads in a warp, i.e. the warp size, has remained 32 across all NVIDIA GPUs. The maximum number of warps available is a function of a multiprocessor's thread-block threshold (which potentially changes dynamically as explained above) and the warp size. The scheduling unit of every multiprocessor maintains a scoreboard that tracks which warps are ready to execute. Among this pool of warps, it dispatches warps in a fine-grained multi-threaded fashion [1]; the exact mechanics of *how* the choice is made is undisclosed and hence we assume nothing in this regard (see Section V).

Once issued, warps execute in SIMD fashion, meaning that all threads in a warp are issued the same instruction. When different threads in a warp want to follow different sides of an `if-then-else` construct (or any conditional construct), **branch divergence** occurs. Branch divergence is handled in hardware by executing every branch path *serially*, one after another, until the *immediate post-dominator* of the branch is reached. Since all threads in a warp see the same instruction stream, inactive threads are masked off temporarily. As soon as all divergent paths have been explored, warp execution continues from the immediate post-dominator; all threads enabled at the branch are re-enabled so that nested branches can be handled. The hardware optimises the case where branch divergence does *not* occur, skipping over the appropriate sequence of instructions as in regular control flow.

The GPU has a separate RAM to the CPU called **device memory**. Data in device memory is accessible to every multiprocessor, and therefore every thread-block can read and write its contents. Access times are extremely slow, in the order of hundreds of cycles, as device memory resides off chip. Since requesting data from global memory is costly, every multiprocessor is equipped with two on-chip caches: a software-managed **shared memory** and a hardware-managed **L1 cache**. Access times of these memories approach those of registers, but only thread-blocks executing on that particular multiprocessor can access them. These caches compete for the same silicon area in that increasing the size of one decreases the size of the other. On a Fermi architecture, 64 KiB of total cache capacity can be configured either as 48 KiB of shared memory with 16 KiB of L1 cache, or 16 KiB of shared memory with 48 KiB of L1 cache. The backing store for all L1 caches is a shared off-chip **L2 cache** whose capacity is a mere 768 KiB on a Fermi GPU.

IV. Impact of Branch Divergence on CFG Construction

Section II described how WCET analyses employing the IPG generally require the CFG as input. In addition there are assumptions that all potential paths through the program are represented in the CFG and that all loops in the CFG are reducible. In this section we show that, for GPU code, conventional algorithms [15]–[17] to construct CFGs violate the first assumption because of the way GPUs implement branch divergence. We present a solution to update the CFG with so-called branch-divergent edges, but demonstrate that this causes the CFG to become irreducible, hence violating the second assumption. We thus describe how the analysis stages must proceed in a specific order. We start by reviewing relevant terminology and notation.

A. Terminology and Notation

A CFG $C = \langle V_C, E_C \rangle$ is a directed graph where V_C are its basic blocks (vertices) and E_C its edges. For any $v \in V_C$: $pred(v) = \{u : (u, v) \in E_C\}$ denotes its set of predecessors and $succ(v) = \{u : (v, u) \in E_C\}$ denotes its set of successors. A natural loop in C has a unique header vertex u, the single entry point to the loop, and one or more loop-back edges from a loop vertex to the header — see [15] for further details. Let $C' = \langle V_C, E'_C \rangle$ be the CFG obtained from C by removing all of its loop-back edges. Then we say a vertex v is a forward branch in C' if, and only if, one of the following holds: v is not a header vertex and $|succ(v)| > 1$; or v is a header vertex, $|succ(v)| > 1$, and every $u \in succ(v)$ belongs to the loop body. A vertex v is a merge vertex provided $|pred(v)| > 1$.

B. Branch-Divergent Edges

Complications arising through branch divergence are best illustrated through an example: Fig. 2 shows a CFG (top left) together with three possible warp executions through this CFG (bottom) on an NVIDIA GPU. We assume for simplicity that a warp only consists of eight threads and have labelled vertices in the warp execution figure according to which threads are active: 1 signals enabled and 0 signals disabled.

Consider the first execution. Initially, all threads branch to b_{12}, but then half of the threads branch to b_{10}, followed immediately by b_{11} (because that is the sole successor of b_{10}). When the last instruction of b_{11} is finished, execution must continue from b_{13}, and not b_{14}, in order to explore the other divergent path from b_{12}. When b_{13} finishes, execution then passes to b_{14} since all sides of the conditional have been explored and b_{14} is the immediate post-dominator of b_{12}. At this point, all threads become active again since all threads were active at b_{12}, and the end of the program is eventually reached at b_{15}.

The other two executions demonstrate similar behaviour, although two properties are particularly noteworthy. First, b_{14} is executed *twice* in both executions: once as the re-convergence vertex for forward branch b_{12} and the other because execution forks that way from b_7. Second, the order in which sides of the branch are executed is implementation

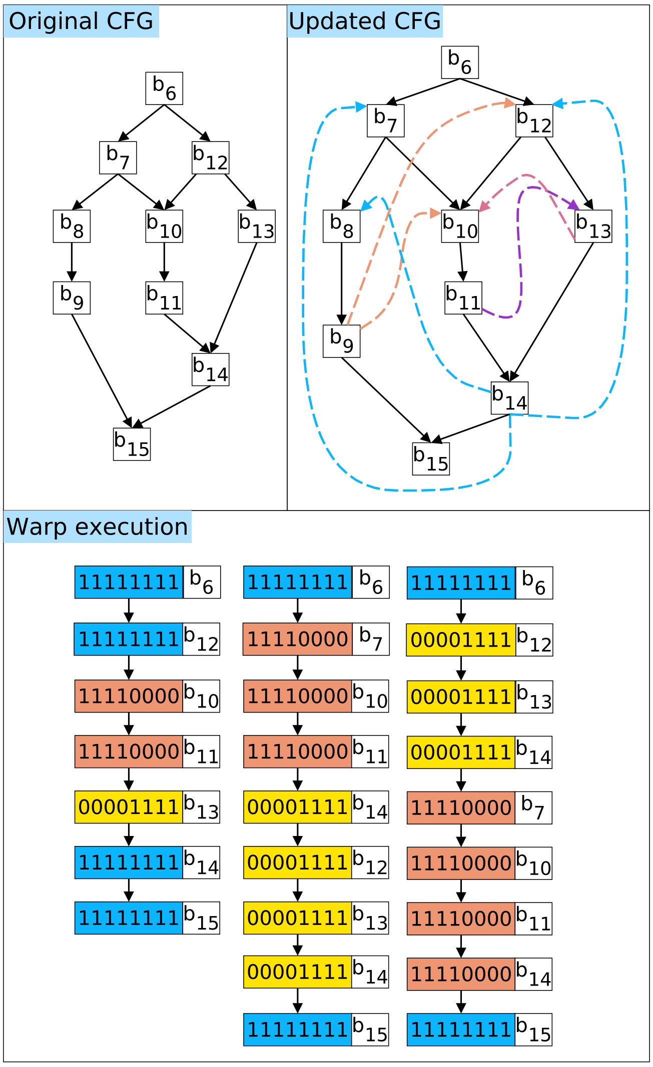


Fig. 2. Example to demonstrate the effect of branch divergence on CFG construction.

defined: in the second execution the edge $b_6 \to b_7$ is followed first, while in the third execution it is the edge $b_6 \to b_{12}$.

As this example establishes, branch divergence alters conventional flow of control: it becomes possible to transfer from a predecessor p of a merge vertex m to a successor $s \neq m$ of a forward branch vertex, even though p only has a single successor m in the CFG. In principle we could solve this problem conservatively by adding extra edges from every predecessor of a merge vertex to every successor of a forward branch vertex. Yet this would yield infeasible paths and could lead to inaccuracies in the execution time estimate. For instance, this solution adds an edge $b_{11} \to b_7$ to the CFG of our running example, although the reader can verify it is impossible for this transfer of control to arise along any branch divergent path.

We instead offer a precise solution grounded on two key observations with respect to a forward branch b, its immediate post-dominator m, and a predecessor p of m.

On the one hand, if only one successor s of b can reach p then s, p must execute on the *same* divergent path; control can then transfer from p to any successor of b except s. For example, in the original CFG of Fig. 2, when execution forks along $b_{12} \to b_{10}$, b_{11} must execute and b_{13} is the *only* vertex where control can transfer after b_{11} if divergence occurred.

One the other hand, if more than one successor of b can reach p then we cannot statically determine which of these successors will lead to execution of p. In principle, control can transfer to *any* successor of b. An example of this property in the original CFG of Fig. 2 is b_{14} with respect to branch b_6: on the one hand, if the path $b_6 \to b_7 \to b_{10} \to b_{11} \to b_{14}$ is followed then b_{12} is executed next; on the other hand, if the path $b_6 \to b_{12} \to b_{10} \to b_{11} \to b_{14}$ is followed then b_7 is executed next.

```
ADD-BRANCH-DIVERGENT-EDGES($C$)
 1  foreach $v \in V_C$ do
 2      $reachable(v) \leftarrow \emptyset$
 3  foreach $v \in V_C$ in reverse post-order do
 4      $reachable(v) \leftarrow \{v\} \cup \left( \bigcup_{p \in pred(v)} reachable(p) \right)$
 5  foreach forward branch $b \in V_C$ do
 6      $m \leftarrow$ immediate post-dominator of $b$
 7      foreach $p \in pred(m)$ do
 8          $newsucc(p) \leftarrow succ(b) \setminus reachable(p)$
 9          if $newsucc(p) \neq \emptyset$ then
10              $E_C \leftarrow E_C \cup \{(p, s) : s \in newsucc(p)\}$
11          else
12              $E_C \leftarrow E_C \cup \{(p, s) : s \in succ(b)\}$
```

Fig. 3. Algorithm to insert branch-divergent edges.

Fig. 3 presents an algorithm to update the edges of a CFG C based on these observations. It uses data-flow analysis [15] to propagate through the CFG which vertices can reach a vertex v, the result of which is stored in $reachable(v)$ (Lines 1–4). Note that a reverse post-order of the CFG is utilised in the second sweep through the vertices so that execution order among vertices is preserved.

The next stage analyses the region in the CFG between a forward branch b and its immediate post-dominator m (Lines 5–6). It deduces, for each predecessor p of m (Line 7), which successors of b currently *cannot* reach p (Line 8), and performs one of two actions. Either a proper subset of b's successors cannot reach p (Line 9), and we add edges to model how control flow potentially branches to one of these locations after the branch-divergent path at p finishes (Line 10). Or, all b's successors can reach p, in which case we cannot infer at compile time which successor led to execution of p, hence we add edges to all of them (Line 12).

Example: The original CFG of Fig. 2 contains three forward branches: b_6, b_7, b_{12}. We will concentrate exclusively on the region (b_6, b_{15}), noting that the successors of $b_6 = \{b_7, b_{12}\}$ and the predecessors of $b_{15} = \{b_9, b_{14}\}$. In this region we compute: $newsucc(b_9) = \{b_7, b_{12}\} \setminus \{b_6, b_7, b_8, b_9\} = \{b_{12}\}$

and hence add the edge $b_9 \to b_{12}$; $newsucc(b_{14}) = \{b_7, b_{12}\} \setminus \{b_6, b_7, b_{10}, b_{11}, b_{12}, b_{13}, b_{14}\} = \emptyset$ and hence add edges $b_{14} \to b_7$ and $b_{14} \to b_{12}$. Performing the same analysis on the regions (b_7, b_{15}) and (b_{12}, b_{14}) leads to the updated CFG pictured in Fig. 2, through which all warp executions can now be traced.

C. Irreducible CFGs

After applying the algorithm of Fig. 3 to a CFG C, a CFG C' enhanced with branch-divergent edges is produced. However, these new edges create irreducible loops in C' even if C were acyclic; for example, in Fig. 2, the original CFG is acyclic whereas the modified CFG has an irreducible loop between b_{11} and b_{13}.

Since the construction of an IPG assumes that the CFG is reducible, the consequence is that the analysis stages and alterations to the CFG must adhere to the following order. Initially the IPG is built using the CFG C that is free from branch-divergent edges; as a consequence, the cycle-inducing edges of the IPG can be detected (see [12]). Next, branch-divergent edges are inserted into the CFG, creating C'. Finally, the edges of the IPG are updated according to whether there are new Ipoint-free paths between Ipoints in C'.

Observe that adding branch-divergent edges after structural analysis of the CFG is applicable to *all* CFG-based WCET analyses. For example, WCET calculations using integer linear programming [14], [26] and loop-bound analysers [13] assume the CFG is reducible.

V. Hybrid WCET Analysis of GPU Code

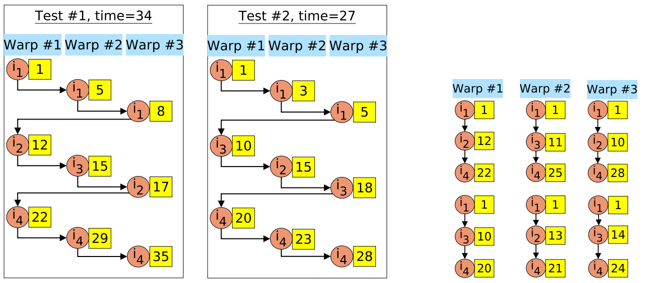
Our overarching aim is to estimate the WCET of GPU code using traces and the IPG, but two additional hurdles remain. First, parallel and concurrent execution on the GPU spawns traces where Ipoints from different warps are *interleaved*, hence blocking trace parsing with the IPG (Section V-A). Second, the WCET calculation performed on the IPG in effect assumes *sequential* execution and ignores how parallelism or concurrency affect timing (Section V-B).

A. Trace Slicing

The motivation for launching multiple thread-blocks is to boost performance by executing the GPU kernel in parallel across multiple multiprocessors. Likewise, maintaining many warps in flight concurrently on a single multiprocessor is motivated by latency hiding since ready warps can be serviced while other warps wait on memory accesses. This parallel and concurrent execution model implies, however, that successive Ipoints written to a trace are not necessarily generated by the same warp on the same multiprocessor. In fact, the interleaving of Ipoints in this fashion must be assumed to be non-deterministic because the exact scheduling policies of thread-blocks to multiprocessors and warps to cores are unknown. In this form the IPG cannot parse these traces because its edges only represent transitions across sequential code constructs.

Our solution is to *slice* traces into a set of **Warp-Specific Traces** (WSTs) whereby only Ipoints generated by a specific warp on a particular multiprocessor are retained in its WST. Since the execution path followed by a warp only flows through the sequential part of code, every WST can then be parsed by the IPG.

Example: Suppose that the program of Fig. 1 has been executed twice (that is, with two different test vectors) on an NVIDIA GPU. Assume for simplicity that there is only a single multiprocessor with three warps scheduled in a round-robin fashion, and no warps execute divergent branches. Traces generated from the example program under these assumptions appear in Fig. 4a: this figure shows which Ipoints each warp triggers, the cycle at which each Ipoint executes, and how execution on the multiprocessor switches between warps.



(a) Traces generated from program in Fig. 1.

(b) Warp-specific traces after trace slicing.

Fig. 4. Example of how traces are processed.

Slicing the traces produces the WSTs of Fig. 4b. Observe how the edges depicting warp interleavings have effectively been stripped away and substituted by transitions between Ipoints in each warp. Conceptually it now appears that warps execute *in parallel* and can all start running at the first clock cycle; to underline this side effect, execution times of Ipoints have been normalised to this baseline. Parsing these WSTs uncovers the following observed WCET of IPG edges:

- $WCET(i_1 \to i_2) = 12$ (test #2, warp #2);
- $WCET(i_1 \to i_3) = 13$ (test #2, warp #3);
- $WCET(i_2 \to i_4) = 18$ (test #1, warp #3);
- $WCET(i_3 \to i_4) = 14$ (test #1, warp #2).

Finally we carry out a WCET calculation on the IPG. There are only two alternative paths in the IPG: $i_1 \to i_2 \xrightarrow{*} i_2 \to i_4$ whose WCET is $12 + 0 + 18 = 30$; or $i_1 \to i_3 \to i_4$ whose WCET is $13 + 14 = 27$. The WCET estimate is thus 30.

B. Accounting for Parallel and Concurrent Execution

Comparing the WCET estimate with the **High Water-Mark Time** (HWMT) of 34 (c.f. Fig. 4a), we observe there is an underestimation. The reason is that this value encapsulates the worst case of an *individual* warp — termed the **warp-specific WCET** in the remainder of the paper — but it ignores two fundamental properties of the execution model. First, when a kernel commences, the scheduler must choose which warp among a pool of warps it will issue to cores; hence, there is always a delay before a final warp within this pool starts executing. Second, there is a tacit assumption that the kernel completes as soon as the warps within this initial pool do so, although generally warps arrive in a series of **waves**, i.e. when the number of thread-blocks launched saturate resources on the multiprocessors (see Section III).

We propose two solutions, both of which are contingent on further analysis of the traces. The first is akin to a dynamic technique in that we simply consider end-to-end behaviour, whereas the second is akin to a hybrid technique in that we create a static analytical model whose actual parameters are obtained from the dynamic profile of the program.

The dynamic technique (c.f. Fig. 5). This approach is grounded on the observation that, on every multiprocessor, there is always a final warp to execute. The idea, therefore, is to analyse the traces generated by a multiprocessor and infer the **worst-case release jitter** of its final warp, defined to be the longest time until the Ipoint s of the final warp executes. Then we conservatively presume that the final warp on every multiprocessor is only released after this delay and that it then goes on to consume the warp-specific WCET. That is:

$$Z_{dynamic} = Z_{warp} + max(\delta_{m_1}, \ldots, \delta_{m_n}) \qquad (1)$$

where Z_{warp} is the warp-specific WCET estimate and, for multiprocessors m_1 through m_n, $\delta_{m_1}, \ldots, \delta_{m_n}$ are the observed worst-case release jitters of their final warps.

Let us apply this equation to our running example, noting from Fig. 4a that the worst-case release jitter of warp #3 (the last scheduled warp in both test cases) is 7 cycles. Once it starts executing, the warp consumes 30 cycles, that is $Z_{dynamic} = 30 + 7 = 37$, which now bounds the HWMT of 34.

It may appear that (1) only accounts for the timing effects of a *single* multiprocessor. However, multiprocessors are parallel processing units, operating largely independently unless they compete for bandwidth to global memory or L2 cache. We argue that the time consumed by these interactions is implicitly included in the warp-specific WCET estimate, because execution times of IPG edges include time spent waiting on these transactions. Furthermore, an accurate worst-case static model of this behaviour is impossible without intricate knowledge of the memory and scheduling policies.

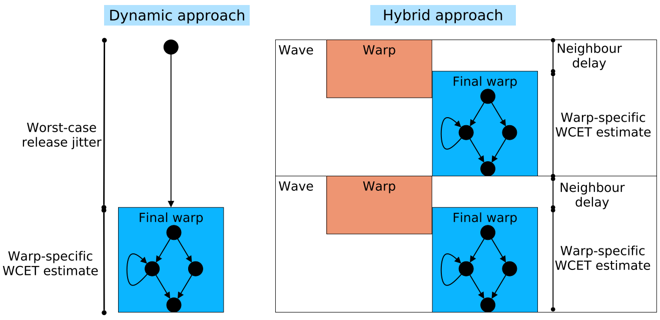


Fig. 5. Incorporating concurrency into the timing analysis.

The hybrid technique (c.f. Fig. 5). The downside of the dynamic approach is that there may always be another test vector or schedule which stalls the final warp even further, and hence the WCET estimate computed by (1) is optimistic. The hybrid approach attempts to resolve this issue by constructing a model of how warps arrive. The crucial observation is that thread-blocks always arrive in waves — as soon as warps within one wave complete, warps spawned by other thread-blocks arrive, until completion. The heart of the model therefore comprises a maximum number of waves, Ω, each wave itself consisting of a maximum number of warps, φ.

This approach also takes a different approach to computing the worst-case release jitter of the final warp. Let the sequence $w_1, w_2, \ldots, w_\varphi$ denote the warps in a wave ordered so that w_i appears before w_j if w_i is scheduled for the first time before w_j is scheduled for the first time. Then we assume that, for every warp w_i with $i > 1$, the start Ipoint of w_i is only written to a trace after a constant-time delay, Δ, with respect to the start Ipoint of its preceding warp w_{i-1}. In effect, every pair of neighbouring warps stall each other by Δ cycles — the delay therefore accumulates and ripples downwards to the final warp.

To combine these values, we assume that, after the worst-case release jitter, the final warp in a wave consumes the warp-specific WCET, and that this pattern repeats for the maximum number of waves. That is:

$$Z_{hybrid} = \Omega \cdot (Z_{warp} + (\varphi - 1) \cdot \Delta) \qquad (2)$$

The values Ω, φ, and Δ are derived by analysing the traces generated on every multiprocessor. The basic idea is that, for a trace T, we maintain a set of warp sets, S_T, where each warp set contains warps observed in a particular wave. Initially S_T contains a single set W. Whenever the parser sees a stream of start Ipoints *without* interleaving exit Ipoints, this indicates that the multiprocessor is starting a new wave of warps, and hence we add the warp identifier to W. However, when an exit Ipoint breaks this sequence, it signifies that the warps in a particular wave are on the verge of completion; at that point, we insert a new empty set W' into S_T in readiness for the next wave of warps. This process is repeated for every trace, and likewise for every multiprocessor. At the end, the maximum number of waves Ω is the maximum size of S_T, while the maximum number of warps φ is the maximum size of W. The value Δ is simply the maximum observed difference between execution times of consecutive start Ipoints in the *same* wave.

We demonstrate this process through Fig. 6, which plots the sequence of start and exit Ipoints observed on a particular multiprocessor horizontally from left to right. Moreover, every Ipoint is subscripted by its warp identifier and, in the case of start Ipoints, time stamps are included. On encountering the sequence of Ipoints $\mathsf{s}_1, \mathsf{s}_2, \mathsf{s}_3$, warps w_1, w_2, w_3 are added to set-1 because there are no exit Ipoints in between. However, Ipoint t_2 signals the end of the first wave and we create the empty set-2; eventually warps w_4 and w_5 become members of this set. In this example, therefore $\Omega = 2$ and $\varphi = max(3, 2) = 3$. Also note that we compute the difference between the time stamps of $(\mathsf{s}_1, \mathsf{s}_2)$, $(\mathsf{s}_2, \mathsf{s}_3)$, $(\mathsf{s}_4, \mathsf{s}_5)$ but *not* $(\mathsf{s}_3, \mathsf{s}_4)$ because w_3, w_4 are not in the same wave; hence $\Delta = max(3, 8, 5) = 8$.

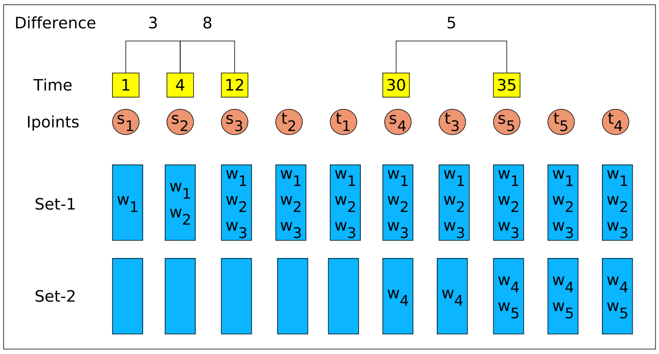


Fig. 6. How the multiprocessor portion of a trace is processed to derive values of parameters in (2).

Application	Description
`BitonicSort`	Parallel algorithm to sort n elements
`BlackScholes`	Computes European pricing options using the Black-Scholes formula
`EigenValues`	Computes all eigenvalues for a square matrix
`Histogram`	Computes a 64-bin histogram
`MatrixMultiply`	Multiples two $n \times m$ matrices
`MatrixTranspose`	Transposes an $n \times m$ matrix
`VectorAdd`	Adds two n-element vectors
`Reduction`	Parallel summation of n elements using a tree-based implementation
`Scan`	Given an array A of n elements, computes an array A' where element $A'[i] = \sum_{j=0}^{i} A[j]$
`ScalarProduct`	Calculates the scalar product of two n-element vectors

TABLE I. Summary of analysed CUDA SDK benchmarks.

Returning to the running example of Fig. 4a, we observe that $\Omega = 1$, $\varphi = 3$, and $\Delta = max(5-1, 8-5, 3-1, 5-3) = 4$. Hence: $Z_{hybrid} = 1 \cdot (30 + 4 \cdot 2) = 38$. Note that this value bounds the HWMT of 34 and that it is more pessimistic than the 37 cycles computed through the dynamic approach; the next section evaluates the differences in more detail using actual GPU kernels.

VI. Evaluation

We have developed a complete WCET toolchain for GPU code based on GPGPU-sim [19] with the aim of obtaining WCET estimates of several kernels in the CUDA SDK [2].

A. GPGPU-sim

GPGPU-sim is a cycle-accurate simulator of NVIDIA hardware, which has been engineered by inspecting NVIDIA patents; it is highly accurate, having been validated against real hardware.

We used a default GPU configuration shipped with the source code that conforms to a Tesla C2050 GPU. This is a Fermi-based architecture which includes the following features: 14 multiprocessors, 32 cores per multiprocessor, 48KiB of shared memory (per multiprocessor), 16KiB of L1 cache (per multiprocessor), 786KiB of L2 cache (shared among multiprocessors) and a clock speed of 1.15 GHz.

GPGPU-sim operates by simulating the PTX (Parallel Thread eXecution) instructions of a kernel. PTX is an assembly language devised by NVIDIA, which is either output as a by-product of compilation from source code, or by an object code disassembler. The latter is hence more accurate as it represents more faithfully what executes on the bare metal, in particular because the CUDA compiler can optimise PTX further; in our experiments, all results are based on disassembled object code.

Some modifications to GPGPU-sim were necessary in order to extract time-stamped traces of execution.[1] We added code to intercept when the first instruction of a basic block is issued to the cores of a multiprocessor, which writes the following to a trace: the address (i.e. the Ipoint ID), the multiprocessor ID, the warp ID, and the cycle time (number of cycles since execution commenced). Hence in our tool, every basic block corresponds to an Ipoint. However, it is important to stress that the analysis works with coarser instrumentation. In addition, because a simulator was deployed to monitor Ipoint execution, Ipoints did not add timing overhead to the application. A full discussion of advantages and disadvantages of these choices is beyond the scope of the paper (see [27]).

B. Benchmarks

We analysed CUDA applications shipped with the CUDA SDK [2]. We selected those for which the application performs meaningful computation (some benchmarks merely illustrate a CUDA feature) and for which it was straightforward to generate a test vector. The specific benchmarks analysed are given in Table I. Note that many applications in the CUDA SDK include several variants of a GPU kernel, moving from a naive implementation to a progressively optimised version, e.g. `MatrixTranspose`. Also, some kernels are called in a sequence to compute a desired output, e.g. `Histogram`. We analysed all kernels in these applications.

C. Experimental Set-Up

Some work was required to sanitise the benchmarks into a form amenable to WCET analysis. First, we stripped the code down to a minimal form that included data transfers and kernel launches, but without compromising the desired functionality. Second, we added a simple random test-vector generator to each benchmark because our hybrid analysis requires a test harness; as all of the benchmarks expect an array of a basic type (e.g. integer or floating point), this was straightforward.

Every program was then compiled using `nvcc` v4.0 (NVIDIA's CUDA compiler) using the default values for thread-block size and number of threads per thread-block as included in the benchmark. After compilation the binary can be executed natively, as GPGPU-sim operates by intercepting calls made by the binary into the CUDA run time. In our experiments, multiple executions (up to eight) of the binary were launched in parallel on a compute cluster that uses nodes with dual 2.66GHz Intel Xeon 5150 cores, 4GiB of RAM, and

[1] Our analysis tools and modifications to GPGPU-sim are publicly available at http://wcet.doc.ic.ac.uk/.

running under RedHat Linux 6.3. We decided to use 1000 test vectors per benchmark, producing 1000 traces that were concatenated together into a single, monolithic trace.

Traces were then split and parsed as detailed in Sections II and V to determine the observed WCET of IPG edges, the maximum execution counts of IPG edges, and to extract the values of parameters in Equation (1) and Equation (2). The warp-specific WCET estimate was computed using an integer linear program derived from the IPG and the data [10].

D. Results

Table II displays the following: the HWMT obtained during testing (Z_{HWMT}); the warp-specific WCET estimate (Z_{warp}); the WCET estimate computed through (1) ($Z_{dynamic}$); the difference between (1) and the HWMT; the WCET estimate computed through (2) (Z_{hybrid}); and the difference between (2) and the HWMT. The units of time in the table are cycles and differences are percentages rounded up to the nearest decimal place. Each row of the table gives the results for a CUDA application — where the application contains multiple kernels, these results have been separated out accordingly.

The most striking observation from these results is the difference between the HWMT and the WCET estimates. Under the assumption that the HWMT *is* the actual WCET, and interpreting the difference as *overestimation*, the dynamic approach is generally much more accurate than the hybrid counterpart: the average overestimation of the former is 102% whereas for the latter it is 796%.

We therefore inspected those kernels where $Z_{hybrid} \gg Z_{dynamic}$ and found that these kernels are always launched with many more thread-blocks than the multiprocessors could service in one chunk. That is, thread-blocks always arrive in multiple waves, and the values obtained from our trace analysis for (2) are sometimes pessimistic.

For example, consider the `MatrixTranspose-1` kernel, which has a small CFG (four basic blocks) and simple control flow properties (one loop but no branches). Through manual inspection of the kernel, we found that it is launched with 1024 thread-blocks and 256 threads per thread-block. On the GPU configuration used in these experiments, each multiprocessor has a maximum capacity of 8 thread-blocks. Therefore, at any one time, there is a maximum of $8 \cdot 14 = 112$ thread-blocks in flight. Assuming a fair scheduling policy where thread-blocks are distributed evenly among multiprocessors, and assuming each multiprocessor pauses until all thread-blocks of a particular wave have completed, the thread-blocks will arrive in $\lceil 1024/112 \rceil = 10$ waves; however, our trace analysis instead computed $\Omega = 26$. This is because we assume that an exit Ipoint followed by a sequence of start Ipoints signals a new wave: if new thread-blocks $b_1, b_2, \ldots, b_n$ of a wave arrive piece by piece and not together in a single batch, it is possible that all warps of thread-block b_i finish before warps of b_{i+1} begin, and we conservatively assume that b_{i+1} is a new wave. The GPU is likely to schedule thread-blocks in this manner because it eagerly allocates unserviced thread-blocks to multiprocessors as soon as other thread-blocks finish. Hence, in future work we will investigate how to better compute the value Ω.

The deduction of the value Δ from traces is similarly hindered by multiple waves. Recall that this value is the maximum difference between the time of start Ipoints of neighbouring warps in the *same* wave. The problem is that some warps from a previous wave, $wave_{old}$, usually remain in flight when a new wave, $wave_{new}$, begins. Hence, warps from $wave_{old}$ are scheduled *in between* warps of $wave_{new}$ and the delay between consecutive warps in $wave_{new}$ can therefore be considerable. In the case of `MatrixTranspose-1`, $\Delta = 1681$, and combined with its maximum number of waves, this already accounts for $26 \cdot 1681 = 43,706$ cycles.

On the other hand, the thread-blocks of `BlackScholes` and `EigenValues` arrive in a single wave, and $\Delta = 1$ because warps are issued to cores immediately in round-robin fashion. In this case, $max(\delta_{m_1}, \ldots \delta_{m_n}) = \Omega \cdot (\Delta \cdot (\varphi - 1))$, and hence $Z_{dynamic} = Z_{hyrbid}$ (c.f. (1) and (2)).

With respect to the warp-specific WCET estimate, we arrive at two conclusions. First, the results confirm that computing Z_{warp} alone is generally not sufficient to bound Z_{HWMT}. Second, when $Z_{warp} > Z_{HWMT}$ holds, the dynamic approach overestimates much more (almost always over 100%) than when the inverse holds: reducing the pessimism in the warp-specific WCET estimate thus remains a key research challenge.

We inspected the source code of kernels in conjunction with the code parts contributing to the warp-specific WCET estimate, e.g. `ScalarProduct`, `Histogram` and `Reduction`, to investigate where execution time is spent. Unsurprisingly, the majority of Z_{warp} is consumed in loops; however, it is noteworthy that each such loop contains a barrier synchronization statement, the semantics of which is to forces all threads in a thread-block to wait at that program point until all threads arrive. Clearly barrier synchronizations are costly because they block warp progress. Our analysis of loops with barriers is pessimistic because it extracts the maximum observed waiting time from the traces and then factors this value by the maximum loop bound. For instance, in the `Reduction-2` kernel, the IPG edge containing execution of the barrier synchronization had a best-case execution time of 33 cycles, an average-case execution time of 58 cycles, and a WCET of 293 cycles. In future work we aim to reduce the pessimism by incorporating the *execution time profiles* of IPG edges into the warp-specific WCET calculation.

The results also show that optimising a kernel can also reduce its HWMT and WCET estimate. For example, the suffix of each `Reduction` kernel indicates an increasingly optimised implementation: `Reduction-4` through `Reduction-6` clearly perform better than `Reduction-1` through `Reduction-3`. Note, however, that both the HWMT and WCET estimates of `Reduction-2` are higher than those of `Reduction-1`, even though the former claims to maintain better warp progress and hence increase parallelism.

Kernel	Z_{HWMT}	Z_{warp}	$Z_{dynamic}$	Difference	Z_{hybrid}	Difference
`BitonicSort-1`	1,045,259	173,548	1,186,941	14%	6,929,575	563%
`BitonicSort-2`	101,582	7,769	107,222	6%	1,438,780	1,316%
`BitonicSort-3`	264,934	39,091	296,152	12%	2,448,819	824%
`BlackScholes`	793,333	408,700	1,080,880	36%	2,138,670	170%
`EigenValues-1`	1,143,429	2,801,330	2,801,337	145%	2,801,337	145%
`EigenValues-2`	1,811,709	4,190,040	4,190,047	131%	4,190,047	131%
`EigenValues-3`	2,576,497	10,292,800	10,292,807	299%	10,292,807	299%
`Histogram-1`	181,164	1,274,430	1,274,469	603%	1,274,469	603%
`Histogram-2`	1,220,518	69,455	1,266,381	4%	2,884,400	136%
`MatrixMultiply`	3,642	4,678	4,680	29%	4,680	29%
`MatrixTranspose-1`	97,303	23,364	117,330	21%	2,661,646	2,635%
`MatrixTranspose-2`	40,621	6,734	44,747	10%	1,314,576	3,136%
`MatrixTranspose-3`	39,671	6,406	43,268	9%	898,864	2,166%
`MatrixTranspose-4`	27,807	6,276	32,282	16%	371,462	1,236%
`Reduction-1`	2,158	4,548	4,555	111%	4,555	111%
`Reduction-2`	2,407	6,442	6,449	168%	6,449	168%
`Reduction-3`	2,625	4,583	4,590	75%	4,590	75%
`Reduction-4`	1,947	3,906	3,913	101%	3,913	101%
`Reduction-5`	1,880	2,580	2,587	38%	2,587	38%
`Reduction-6`	1,645	2,196	2,203	34%	2,203	34%
`ScalarProduct`	107,625	684,002	744,425	591%	1,378,438	1,181%
`Scan-1`	98,838	52,350	146,603	48%	4,097,887	4,064%
`Scan-2`	34,116	50,886	50,893	49%	50,893	49%
`Scan-3`	46,328	2,417	47,940	3%	373,170	705%
`VectorAdd`	656	652	659	< 1%	659	< 1%

TABLE II. Analysis results for benchmarks. All execution times are in cycles.

VII. Conclusions

This paper extended a previous hybrid technique to estimate the WCET of sequential code so that it now targets GPU applications running on NVIDIA hardware. We proposed two ways in which to incorporate the effect of concurrency into the timing model: one is a pure dynamic technique using measurements alone, while the other is a hybrid technique in that it computes a value through a static analytical model whose parameter values are derived from measurements. By analysing several GPU kernels from the CUDA SDK, our results show that the former method is much more accurate: indeed, our principal conclusion is that how concurrency is integrated into the timing model largely determines the degree of accuracy. Future work will investigate how to automatically diagnose performance bottlenecks in GPU applications using our performance model, and the applicability of our framework to GPUs manufactured by other companies.

References

[1] J. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. Morgan Kaufmann Publishers, 2011.

[2] NVIDIA, "The CUDA standard development kit, version 4.0," https://developer.nvidia.com/cuda-downloads, 2012.

[3] Khronos OpenCL Working Group, "The OpenCL specification, version 1.1," 2011.

[4] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan, "Liszt: a domain specific language for building portable mesh-based PDE solvers," in *SC*, 2011.

[5] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Computing Surveys*, vol. 43, no. 4, pp. 1–44, 2011.

[6] B. Lisper, "Towards parallel programming models for predictability," in *Workshop on WCET Analysis*, 2012.

[7] NVIDIA, http://www.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 Computer Systems*, vol. 7, no. 3, pp. 36:1–36:53, 2008.

[9] G. Bernat, A. Colin, and S. M. Petters, "WCET analysis of probabilistic hard real-time systems," in *RTSS*, 2002.

[10] A. Betts, "Hybrid Measurement-Based WCET Analysis using Instrumentation Point Graphs," Ph.D. dissertation, University of York, 2008.

[11] A. Colin and S. M. Petters, "Experimental Evaluation of Code Properties for WCET Analysis," in *RTSS*, 2003.

[12] A. Betts and G. Bernat, "Identifying Irreducible Loops in the Instrumentation Point Graph," *Journal of Systems Architecture*, vol. 3063, no. 2, pp. 78–90, 2010.

[13] A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper, "Loop Bound Analysis based on a Combination of Program Slicing, Abstract Interpretation, and Invariant Analysis," in *Workshop on WCET Analysis*, 2007.

[14] P. Puschner and A. V. Schedl, "Computing Maximum Task Execution Times - A Graph-Based Approach," *Real-Time Systems*, vol. 13, no. 1, pp. 67–91, 1997.

[15] A. Aho, R. Sethi, M. S. Lam, and J. Ullman, *Compilers: Principles, Techniques and Tools*, 2nd ed. Addison-Wesley, 2006.

[16] H. Theiling, "Control Flow Graphs for Real-Time System Analysis: Reconstruction from Binary Executables and Usage in ILP-Based Path Analysis," Ph.D. dissertation, Universität des Saarlandes, 2002.

[17] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[18] G. Ramalingam, "On Loops, Dominators, and Dominance Frontiers," *TOPLAS*, vol. 24, no. 5, pp. 455–490, 2002.

[19] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *ISPASS*, 2009.

[20] Rapita Systems Ltd., http://www.rapitasystems.com/.

[21] The Nexus 5001$^{\text{TM}}$Forum, http://www.nexus5001.org.

[22] S. M. Petters, "Comparison of trace generation methods for measurement based WCET analysis," in *Workshop on WCET Analysis*, 2003.

[23] J. Wegener and F. Mueller, "A Comparison of Static Analysis and Evolutionary Testing for the Verification of Timing Constraints," *Real-Time Systems*, vol. 21, no. 3, pp. 241–268, 2001.

[24] R. E. Tarjan, "Testing Flow Graph Reducibility," *Journal of Computer and System Sciences*, vol. 9, pp. 355–365, 1974.

[25] A. Betts and G. Bernat, "Tree-Based WCET Analysis on Instrumentation Point Graphs," in *ISORC*, 2006.

[26] A. Betts, "Reducing the Size of the Constraint Model in Implicit Path Enumeration using Super Blocks," in *RTSS*, 2012.

[27] A. Betts and A. Marref, "WCET Analysis of Component-Based Systems using Timing Traces," in *ICECCS*, 2011.