A Compiler Phase to Optimally Split GPU Wavefronts for Safety-Critical Systems

Artem Klashtorny
Electrical and Computer Engineering
University of Waterloo
Waterloo, Canada
artem.klashtorny@uwaterloo.ca

Mahesh Tripunitara
Electrical and Computer Engineering
University of Waterloo
Waterloo, Canada
tripunit@uwaterloo.ca

Hiren Patel
Electrical and Computer Engineering
University of Waterloo
Waterloo, Canada
hiren.patel@uwaterloo.ca

***Abstract*—We present a compiler phase for GPUs enabled with predictable wavefront splitting (PWSG) that implements an optimal algorithm to split diverging GPU wavefronts into separate scheduleable entities. This algorithm selects branches in the GPU kernel that guarantee the lowest worst-case execution time (WCET) for the kernel. We implement our algorithm in a compiler flow for an AMD GPU, and we deploy the resulting binary on a gem5 micro-architectural implementation of the AMD GCN3 GPU. We evaluate our implementation on an extensive set of synthetic benchmarks. Our experiments show that by automatically selecting points in the GPU kernel to split wavefronts, we are able to reduce the WCET ranging from 34% to 52% reduction compared to five alternative approaches.**

***Index Terms*—parallel architectures, real-time systems, SIMD processors, compilers, algorithms**

I. Introduction

There is growing interest in using graphics processing units (GPUs) for safety-critical systems. This is because GPUs offer significant performance improvements for massively parallel workloads. An example of a safety-critical systems application that can benefit from using GPUs is autonomous driving. GPUs can be used in autonomous driving systems to run speed limit recognition, blind spot identification, parking assistance, and active crash prevention. However, adopting commercial-off-the-shelf (COTS) GPUs in safety-critical systems is a challenge as we explain next.

Applications that use GPUs for safety-critical systems typically undergo a certification process to ensure that the worst-case execution time (WCET) of the application is never exceeded. Using active crash prevention as an example, the brakes must always be applied within a certain amount of time to avoid crashes. Computing the WCET requires detailed knowledge of the GPU hardware, the software driver, the remaining software stack, and its interaction with the hardware. However, obtaining this information for COTS GPUs, such as the NVIDIA Jetson SoC and the Tesla Autopilot SoC is difficult. This is because their implementations are proprietary and often closed source [1].

In response, a recent research effort proposed a predictable GPU architecture with wavefront splitting (PWSG) [2] for safety-critical systems built on top of the AMD GCN3 architecture that is open-source [3]. PWSG's design exploits a performance optimization technique while lowering the WCET. PWSG accomplishes this by making the following three contributions. (1) PWSG exploits a form of predictable wavefront splitting to improve the performance of divergent branches, and reduce the WCET of the kernel. (2) Two instructions are added to the instruction-set architecture to allow programmers to explicitly annotate branches in the kernel to split wavefronts, and points to merge wavefronts. This allows analysis tools to compute the WCET of the kernel and lower the WCET estimates. (3) An extended WCET analysis with support for predictable wavefront splitting. We find PWSG to be an interesting GPU architecture for safety-critical systems. However, we found it difficult to exploit wavefront splitting effectively while lowering the WCET. This is because PWSG requires manual annotation of branch instructions for wavefront splitting. In our experience, for complex GPU kernels, determining where to make these annotations is cumbersome and often results in large WCET estimates.

In this work, we address this problem by developing a compiler phase for PWSG to optimally split the GPU kernel's wavefronts to minimize the WCET. Our compiler phase implements an optimal algorithm that identifies branches in the GPU kernel that the wavefront can split on. We extend PWSG's compiler flow with this compiler phase to automate the entire process of annotating and generating the GPU kernel executable.

Our main contributions to this work are as follows.

1) We design an optimal algorithm that identifies branches at which to split the wavefronts. This algorithm relies on identifying a recurrence for the optimal choice of branches to annotate to split wavefronts on.
2) We implement the algorithm in a compiler phase that analyzes the control-flow graph and automatically inserts the necessary annotations. We prove the algorithm's correctness and optimality.
3) We integrate our compiler phase with the PWSG compiler flow and empirically evaluate it on a large set of synthetic benchmarks. Our results show that our algorithm is indeed optimal, and we see a reduction in WCET ranging from 34% to 52% when compared to five alternative approaches.

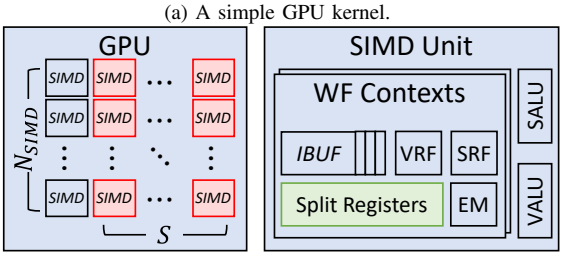
979-8-3503-4859-0/DATE24/© 2024 EDAA

II. Background

We describe the relevant background in understanding a GPU architecture with predictable wavefront splitting.

```
1 void kernel() {
2   asm(split);
3   if (A[wid] > 0)
4     A[wid] += 1;
5   else
6     A[wid] += 2;
7   asm(merge);
8 }
9 int W = 4;
10 launch(kernel, W);
```

(a) A simple GPU kernel.



(b) PWSG hardware model. Dedicated SIMD units are in red and splitting registers are in green.

Fig. 1: GPU system model.

Programming Model. GPUs execute multi-threaded programs known as kernels, where each thread is known as a work-item. Figure 1a shows a simple kernel where every work-item increments the data in a vector `A` at index wid, the work-item identifier. Work-items execute kernel instructions in parallel on distinct data elements; this pattern is called single instruction, multiple data (SIMD) [4]. When launching the kernel, the GPU programmer specifies the number of work-items that execute the kernel, as shown with `int W` in Figure 1a. If the GPU does not have enough execution resources to execute all work-items simultaneously, the GPU divides them into groups of work-items based on the number of ALUs. Such a group is called a wavefront.

Hardware Model. Figure 1b shows the architecture of PWSG: a GPU enabled with predictable wavefront splitting [2]. Note that there are two key differences between a traditional GPU and PWSG: (1) PWSG reserves execution units and wavefront contexts for wavefront splitting, and (2) PWSG introduces `merge` and `split` instructions to statically identify points in the kernel to split wavefronts, as shown in Lines 2 and 7 of Figure 1a. PWSG consists of N_{SIMD} SIMD vector execution units as shown on the left-hand side of Figure 1b. Each SIMD unit contains a vector ALU (VALU) that operates on N_{ALU} data elements in SIMD manner. Hence, one SIMD unit enables SIMD execution of one wavefront at a time. Each SIMD unit also contains a set of wavefront contexts. A wavefront context maintains the state of the wavefront via a vector register file (VRF), a scalar register file (SRF), and an instruction buffer (IBUF). A special register known as an execute mask (EM) specifies whether a work-item is active or dormant. An active work-item executes the instruction, while a dormant work-item executes a `NOP`. If the wavefront executing on the SIMD unit encounters a stall condition, the PWSG can switch to a wavefront stored in another wavefront context to improve performance. PWSG also contains scalar ALUs to execute instructions that are not vector instructions, such as branch instructions and synchronization barriers. The right-hand side of Figure 1b illustrates the detailed architecture of a SIMD unit.

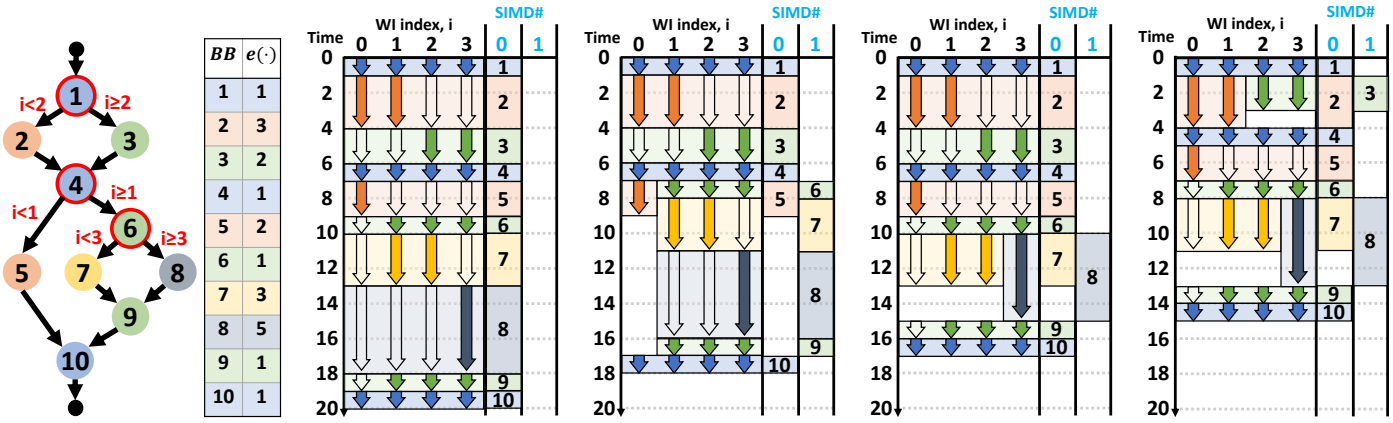
Wavefront Splitting in PWSG. Wavefront splitting is a performance optimization technique to address branch divergence [5]–[8]. Branch divergence occurs when a GPU executes a conditional branch instruction, and the outcome of the condition's evaluation is different across work-items in a wavefront. This causes a subset of the work-items in the wavefront to go down the `true` path for the condition while the other work-items go down the `false` path. Since work-items in a wavefront must execute in lockstep, GPU serializes the execution of the two branch paths using the EM. Consider the program illustrated in Figure 2a using a control-flow graph (CFG). The nodes are basic blocks (b_i) that correspond to sequences of instructions in the kernel between branch instructions. Nodes b_1, b_4, and b_6 branch out to two different nodes based on the value of the work-item index, wid. Figure 2b shows the execution with branch divergence. At time 2, notice that work-items with index 0 and 1 remain active (solid colored arrows) while 2 and 3 are dormant (empty arrows). This is because the branch in b_1 caused work-items 0 and 1 to enter the `true` path. At time 4, the other work-items execute while 0 and 1 are dormant since they completed their execution. Such diverging execution continues for b_4 and b_6. Branch divergence reduces average-case performance [9] and incurs serialized execution for the WCET [2]. PWSG is also a GPU; thus it experiences the same challenges with branch divergence [2].

PWSG enables wavefront splitting by reserving SIMD units to guarantee parallel execution of split wavefronts [2]. This allows PWSG to execute both branching paths for a wavefront in parallel. In Figure 1b, we show the reserved SIMD unit resources in red. PWSG promotes a design parameter $\mathcal{S}$ that indicates the number of units into which any wavefront can be split and executed in parallel. This is done by reserving a subset of SIMD units specifically to execute the split wavefronts. In general, PWSG requires $\mathcal{S} \cdot N_{SIMD}$ reserved SIMD units in total based on the design parameter $\mathcal{S}$. For example, when $\mathcal{S}$ is 1, there need to be N_{SIMD} SIMD units reserved. PWSG also extends the wavefront context with some special registers, highlighted in green, that keep track of relationships between split wavefronts for when they need to merge. PWSG requires GPU programmers to annotate the kernel with `split` and `merge` instructions to select split and merge points. These points indicate where to split wavefronts and merge them.

III. Motivation: Selecting Split and Merge Points

Recall that PWSG reserves $\mathcal{S} \cdot N_{SIMD}$ SIMD units for executing wavefronts that are split. However, PWSG requires the GPU programmer to manually annotate in the kernel where to split and merge the wavefront by using `split` and `merge` instructions [2]. We find deciding where to split and merge to provide the lowest WCET to be a challenge.

Selecting branches to split. We revisit the example shown in Figure 2a. The table beside the CFG indicates the WCETs



(a) Kernel CFG and WCETs for each basic block. (b) Execution timeline without splitting. (c) Execution timeline when splitting at b_4. (d) Execution timeline when splitting at b_6. (e) Execution timeline when splitting at b_1 and b_6.

Fig. 2: Example kernel CFG and the impact of split point choice on WCET.

for each basic block, denoted by $e(b_i)$. We assume that $e(b_i)$ is an input to our compiler phase obtained by either static WCET analysis [10] or measurement-based analysis [11]. Let us assume that $\mathcal{S}$ is 1; thus, PWSG has reserved SIMD units to split a wavefront once. Further, suppose that work-items at b_4 and b_6 diverge. As noted earlier, Figure 2b illustrates the scenario without any wavefront splitting where the diverging work-items serialize their execution. Thus, the WCET of the wavefront executing the kernel is 20 time units. As alternatives, the execution timelines in Figures 2c and 2d show the impact of selecting either b_4 or b_6 to split on, respectively. Splitting at b_4 results in a WCET of 18, and at b_6, the WCET is 17. For this example, annotating the kernel to split wavefronts at b_6 yields the lowest WCET. The main reason the WCET gets reduced when splitting at a branch is because PWSG guarantees parallel execution of both paths of the branch. Thus, the WCET of the branch is the maximum of the two paths. We find that for larger CFGs with multiple nested branches manually selecting branches to split the wavefront on is not practical. Therefore, an automated method to select branches must be developed.

Reusing SIMD units by merging wavefronts. After a wavefront splits, PWSG allows the wavefronts to merge together [2] by requiring the GPU programmer to insert `merge` instructions in the kernel. This typically happens when two branch paths complete their execution and arrive at the join point in the CFG. For example, b_9 is the join point for the branch at b_6. Suppose we annotate b_9 with `merge`. Then, PWSG merges the split wavefront with the original wavefront. The main advantage of merging wavefronts is that the reserved SIMD unit is vacant, which allows other wavefronts to reuse the SIMD unit. We show the benefit of reusing a SIMD unit with the help of Figure 2e. Suppose that the branch at b_1 is marked to split, and b_4 is marked to merge the split wavefronts executing b_2 and b_3. The same SIMD unit can be reused to split the branch at b_4 or b_6. We observe that by selecting branches b_1 and b_6 to split, and b_4 and b_9 to merge on, we obtain a WCET of 15. This is lower than any of the other alternatives.

Challenge: Selecting branches to split on to obtain the lowest WCET while reusing SIMD units. For a simple kernel as in Figure 2, manual inspection may give the optimal assignment of branches to split on. However, in our experience, making these decisions for larger kernels is difficult. Therefore, we require an approach to select branches to split on to obtain the lowest WCET for the kernel while reusing the SIMD units whenever possible. We address both of these challenges.

IV. Compiler Phase

We extend PWSG's compiler flow with our compiler phase and illustrate it in Figure 3. Note that blue boxes in this diagram are new steps that we add, while yellow boxes are inputs and the grey box is for the existing PWSG implementation. The tool flow starts by using PWSG's compiler flow to extract the WCET of basic blocks, $e(b_i)$, which we use to create a detailed CFG. It then feeds this CFG to our split point solver, in which our selection algorithm determines the optimal split points given a value of $\mathcal{S}$. Finally, the tool flow automatically annotates the kernel at the chosen split points.

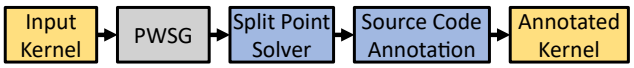
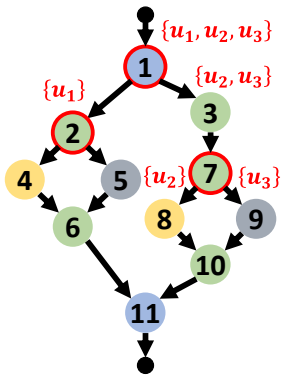


Fig. 3: Compiler phase tool flow.

A. Algorithm

Underlying our algorithm, and a proof for its optimality, is the recurrence in Equation (1) for the optimum, i.e., lowest, WCET we are able to achieve. We adopt the following notation. The basic blocks in the CFG comprise the set $B = \{b_1, \ldots, b_n\}$ while $D \subset B$ is the set of branch nodes. We represent as $t(b_i, u)$ the optimum WCET for the CFG rooted at b_i given u SIMD units; thus, we seek $t(b_1, \mathcal{S}+1)$, where b_1 is the root of the CFG. Note that $\mathcal{S}+1$ is the number of SIMD units that can execute split wavefronts in parallel, referring to the existing SIMD unit and the $\mathcal{S}$ reserved SIMD units. We abuse notation slightly and use $t(C, u)$, where C is a set of b_i values, to refer



BB	e(·)
1	1
2	3
3	1
4	1
5	2
6	1
7	3
8	5
9	6
10	1
11	0

(a) Assignment of SIMD units.

BB	1	2	3
1	24	17	12
2	6	5	5
3	1	1	1
4	1	1	1
5	2	2	2
6	1	1	1
7	14	9	9
8	5	5	5
9	6	6	6
10	1	1	1
11	0	0	0

(b) The table $t[\cdot]$.

BB	1	2	3
1	$\langle(d),1,1\rangle$	$\langle(s),1,1\rangle$	$\langle(s),1,2\rangle$
2	$\langle(d),1,1\rangle$	$\langle(s),1,1\rangle$	$\langle(s),1,2\rangle$
7	$\langle(d),1,1\rangle$	$\langle(s),1,1\rangle$	$\langle(s),1,2\rangle$

(c) The table $n[\cdot]$, only showing entries for branch nodes.

Fig. 4: An example kernel and corresponding tables $t[\cdot]$ and $n[\cdot]$ computed by Algorithm 1 and the consequent assignment of SIMD units. The cells in green highlight the entries that pertain to an optimal solution.

to $\sum_{b\in C} t(b,u)$. We assume a function $e\colon B \to \mathbb{R}^+$, which is the execution time of each b_i. And finally, we use l_i to represent the set of basic blocks contained in the left (i.e., true) branch of b_i that is not contained in the l_k or r_k of any other b_k, and r_i of the right branch; if b_i is not a branch node, then $l_i = r_i = \emptyset$. For example, in Figure 4a, $l_1 = \{2,6\}, r_1 = \{3,7,10\}$, and $l_7 = \{8\}, r_7 = \{9\}$.

$$t(b_i,u) = \begin{cases} e(b_i), & \text{if } b_i \notin D & \text{(1a)} \\ e(b_i) + t(l_i \cup r_i, 1), & \text{if } u = 1 \text{ and } b_i \in D & \text{(1b)} \\ e(b_i) + M, & \text{otherwise} & \text{(1c)} \end{cases}$$

where

$$M = \min\Big\{ \max_{1\le\delta<u} \{t(l_i,\delta), t(r_i,u-\delta)\}, \; t(l_i,u) + t(r_i,1), \; t(l_i,1) + t(r_i,u) \Big\} \quad (2)$$

Theorem 1 (Optimality): The recurrence for $t(b_i,u)$ is correct, i.e., $t(b_1, \mathcal{S}+1)$ yields the minimum WCET for a CFG rooted at b_1, given $\mathcal{S}$ reserved SIMD units.

Proof: By structural induction on the CFG. For the base case, the CFG has one node only, and the recurrence specifies that $t(b_1,u) = e(b_1)$ for all $u \ge 1$, which is of course correct. For the step, if $u = 1$, the entire CFG must be executed using a single SIMD unit only; this corresponds to the second case of the recurrence, i.e., $e(b_i) + t(l_i \cup r_i, 1)$. Otherwise, we either perform a split by allocating, for some δ that satisfies $1 \le \delta < u$, δ units to the left branch and $u-\delta$ to the right, or not split, in which case we allocate all of the u SIMD units to one of the branches. Amongst these options, we choose the one that minimizes the WCET. ■

Algorithm 1 Dynamic programming split hardware allocation

```
1:  procedure FillTables(B, u)
2:      Allocate table t of size |B| × u
3:      Allocate table n of size |B| × u
4:      for b_i ∈ B \ D do
5:          for j from 1 to u do
6:              t[b_i, j] ← e(b_i)
7:      for b_i in B in reverse topological order do
8:          t[b_i, 1] ← e(b_i) + Σ_{k∈l_i∪r_i} t[b_k, 1]
9:          n[b_i, 1] ← ⟨(d), 1, 1⟩
10:     for j from 2 to u do
11:         for b_i in B in reverse topological order do
12:             m ← e(b_i) + Σ_{k∈l_i} t[b_k, 1] + Σ_{k∈r_i} t[b_k, j]
13:             n[b_i, j] ← ⟨(d), 1, j⟩
14:             m_r ← e(b_i) + Σ_{k∈l_i} t[b_k, j] + Σ_{k∈r_i} t[b_k, 1]
15:             if m_r < m then
16:                 m ← m_r
17:                 n[b_i, j] ← ⟨(d), j, 1⟩
18:             for δ from 1 to j − 1 do
19:                 p_l ← Σ_{k∈l_i} t[b_k, δ]
20:                 p_r ← Σ_{k∈r_i} t[b_k, j − δ]
21:                 if e(b_i) + max{p_l, p_r} < m then
22:                     t[b_i, j] ← e(b_i) + max{p_l, p_r}
23:                     n[b_i, j] ← ⟨(s), δ, j − δ⟩
24:                     m ← t[b_i, j]
25:     return
```

Given the recurrence for $t(b_i,u)$, we can write a companion recurrence, shown in Equation (3). This recurrence specifies what we denote as $n(b_i,u)$, which is a triple of $\{(s)\text{plit}, (d)\text{o not split}\} \times \mathbb{Z}^+ \times \mathbb{Z}^+$. The first component of $n(b_i,u)$ identifies whether or not we split at node b_i, given u SIMD units. The other two components identify the manner in which we apportion the u SIMD units to the two subtrees of b_i. As the recurrence expresses, the value of $n(b_i,u)$ is based on which of the cases in the recurrence for $t(b_i,u)$ applies, i.e., minimizes the WCET.

$$n(b_i,u) = \begin{cases} \big\langle (d), 0, 0 \big\rangle, & \text{if } b_i \notin D & \text{(3a)} \\ \big\langle (d), 1, 1 \big\rangle, & \text{if } u = 1 \text{ and } b_i \in D & \text{(3b)} \\ q, & \text{otherwise} & \text{(3c)} \end{cases}$$

where the choice $q \in Q$ minimizes the WCET and

$$Q = \Big\{ \big\langle (s), \delta, u-\delta \big\rangle, \big\langle (d), u, 1 \big\rangle, \big\langle (d), 1, u \big\rangle \Big\} \quad (4)$$

For example, if $n(b_i,5) = \langle(s),2,3\rangle$, this means that we split at b_i while allocating 2 of the 5 SIMD units to the left branch, and 3 to the right. As another example, if $n(b_i,5) = \langle(d),5,1\rangle$, this means that we do not split at b_i, and allocate all 5 SIMD units to the left branch.

Realizing the recurrence for t and/or n top-down, i.e., directly as expressed by the recurrences, would yield an exponential-time algorithm. However, bottom-up, using the dynamic programming algorithm we show as Algorithm 1 is polynomial-time. The algorithm is invoked as FillTables$(\{b_1,\ldots,b_n\}, \mathcal{S}+1)$.

The algorithm allocates and populates two tables, $t[\cdot]$ and $n[\cdot]$, each of size $|B| \times u$, where B is its first argument and u

its second. Each $t[b_i, j]$ corresponds to $t(b_i, j)$, and each $n[b_i, j]$ corresponds to $n(b_i, j)$. The algorithm realizes exactly the recurrence for t from above, and the corresponding companion recurrence for n. The algorithm runs in time $O(n \cdot u^2)$. Thus, its running time is polynomial in the size of the CFG and the number of SIMD units. As the latter is upper-bounded by n, the algorithm is polynomial-time.

In Figures 4b and 4c, we show the tables $t[\cdot]$ and $n[\cdot]$ that are computed by our algorithm for the CFG and corresponding $e(\cdot)$ values in Figure 4a, given $\mathcal{S} = 2$. The entry to the top right of the table for $t[\cdot]$ is our optimal WCET, 12 in this example. The corresponding entry in the $n[\cdot]$ table is $\langle (s), 1, 2 \rangle$. This tells us that we should indeed split at b_1, while allocating 1 SIMD unit to the left branch, and 2 to the right. Thus, back in the table for $t[\cdot]$, we need to consult the rows that correspond to the column for 1 SIMD unit for rows 2 and 6 for the left branch, and the column for 2 SIMD units for the rows 3, 7, and 10 for the right branch. As b_7 is a branch node, we then consult $n[7, 2]$ to learn that we should split at that node, while allocating 1 SIMD unit each to the left and right branches.

A final detail regards the manner in which we determine which of $\mathcal{S} + 1$ units we actually assign to each $b_i \in B$. We do this with a straightforward algorithm that goes top-down the CFG while consulting the entries in the $n[\cdot]$ table. We annotate some of the edges in Figure 4a showing the SIMD units mapped to the branches. As we have 3 SIMD units as input, we adopt the set $\{u_1, u_2, u_3\}$ to represent them. As the table for $n[\cdot]$ says that we split at node b_1 while allocating 1 SIMD unit to the left and 2 to the right, we correspondingly partition our set as Figure 4a shows — $\{u_1\}$ to the left and $\{u_2, u_3\}$ to the right. At branch node b_7 in the right subtree, our table for $n[\cdot]$ again says that we split, while allocating 1 SIMD unit each to the left and right branches, and therefore we partition our set into two, $\{u_2\}$ and $\{u_3\}$. After we run this top-down algorithm, if a node is annotated with a set of SIMD units $\{u_1, \ldots, u_i\}$, then any of $u_1, \ldots, u_i$ SIMD units may be allocated to that node during execution.

V. Evaluation

We evaluate our compiler phase on a publicly available implementation of PWSG [2], [3]. This implementation of PWSG uses the gem5 simulator to implement an AMD GPU with the micro-architectural extensions for predictable wavefront splitting [2]. This implementation allows us to assess the impact of our compiler phase on the WCET. We refer to all these approaches to select branches in the kernel to split on as approaches for selecting split points.

Algorithms. We compare an implementation of our algorithm shown in Algorithm 1 against five other algorithms. We allocate a timeout of five minutes for each algorithm to discover solutions. Our baseline algorithm is no splitting; thus, zero branches are chosen as split points. The second algorithm uses brute force to select points in the kernel to split wavefronts. The third is a naive algorithm that first computes the reduction in WCET of splitting at each branch in isolation. Then, the algorithm selects branches that deliver the greatest reduction. The fourth algorithm randomly selects a set of $\mathcal{S}$ branches to act as split points. The final algorithm is dynamic waveform splitting (DWS) that splits wavefronts at runtime.

Benchmarks. We use an extensive set of synthetic benchmarks to evaluate our compiler phase. Synthetic benchmarks play an important role in exercising a variety of diverse branching patterns that allow us to compare our approach against other approaches.

Synthetic benchmarks. Synthetic benchmarks vary the overall nesting depth of branches in the kernel, the number of instructions in each path, and the number of branches in sequence. All three of these factors impact the number of possible branches to select as split points. Nesting depth refers to the maximum number of nested conditional statement in a GPU kernel. We have five nesting depths from two to six. PWSG groups 64 work-items into each wavefront; hence, a nesting depth of six enables maximum branch divergence for a kernel.

For each nesting depth, we generate five benchmarks, labelled a–e in Figures 5 and 6. These refer to the maximum number of branches for each benchmark that are in sequence with each other, from two to six. For each branching path, we randomize the number of instructions executed to create a variety of distributions of $e(b_i)$ values. We also randomize the number of branches in sequence with each other. These sequences enable wavefront splitting to leverage reuse of dedicated hardware. They allow us to assess the abilities of each algorithm to optimize the split hardware allocation.

Results. To gather results, we pass each of the synthetic benchmarks through each of the split point selection algorithms, assuming a PWSG architecture with $\mathcal{S} = 2$. Larger $\mathcal{S}$ values are possible, and provide more opportunities to split, making it most beneficial to workloads with deep nesting. Using this configuration, we compute the analytical WCET for a wavefront executing the kernel. Figure 5 shows these WCET values for the synthetic benchmarks, and Figure 6 shows the runtime of each algorithm.

The WCET results demonstrate the optimality of the dynamic programming approach to split point selection. It provides an average 34% WCET reduction compared to the naive approach, 36% compared to DWS, 48% compared to the brute force and random approaches, and 52% compared to no splitting. The results also show that the brute force approach can be useful for small benchmarks with nesting depth no greater than three. However, the runtime of the algorithm is exponential; for larger numbers of branches it reaches a timeout condition and does not find a solution to the problem.

VI. Related Work

There are several recent efforts that build on each other and propose wavefront splitting as a solution to address branch divergence [2], [5]–[8]. The first such solution was dynamic warp subdivision [5], which split wavefronts at runtime and interleaved their execution. This work was followed up with simultaneous branch and warp interweaving [7]. In this work, the authors proposed allowing split wavefronts to execute

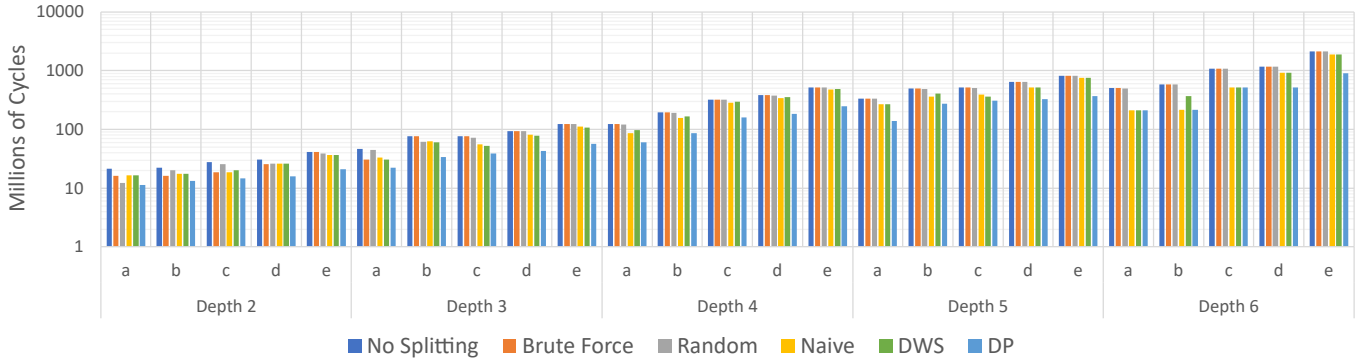


Fig. 5: Computed WCET for synthetic benchmarks given split points selected by each algorithm.

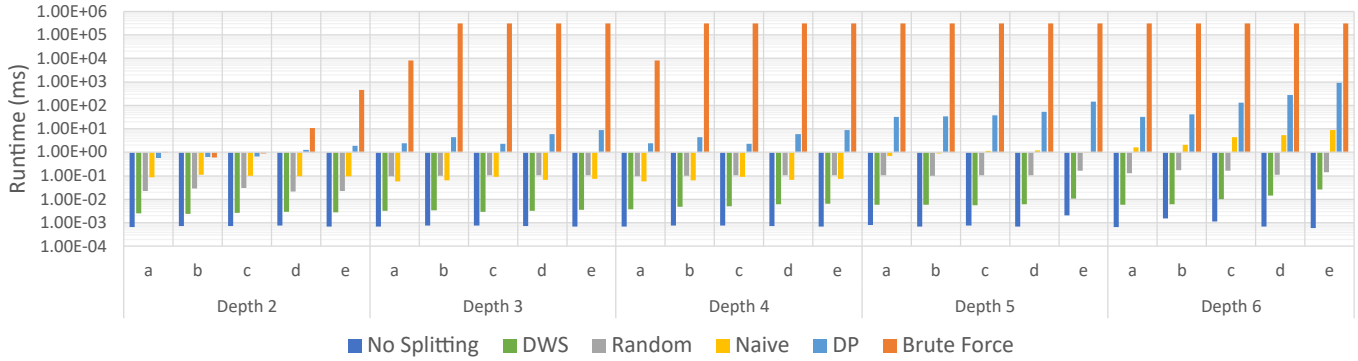


Fig. 6: Algorithm runtimes for each synthetic benchmark.

simultaneously in parallel on shared hardware rather than interleaved. The third solution proposed dual-path execution model [6], which added hardware structures to support merging split wavefronts as soon as possible at the join point of the branch. More recently, subwarp interleaving [8] was proposed, which included independent scheduling information for each work-item, allowing for each wavefront to be split more times. These wavefront splitting works were limited in their benefit to the WCET, because they were unable to guarantee parallel execution of split wavefronts and did not provide any static information of which branches are chosen as split points. Klashtorny et al. proposed predictable wavefront splitting [2], which added dedicated SIMD hardware for splitting to guarantee parallel execution of split wavefronts and enabled the GPU programmer to statically annotate split and merge points in the kernel. These changes to the architecture allow for WCET analysis of a GPU kernel executed using wavefront splitting.

VII. Conclusions

In this work, we present a compiler phase for the PWSG GPU architecture that was designed for safety-critical systems. Our compiler phase optimally selects points in the GPU kernel to split wavefronts to obtain the lowest WCET. The key impact of our compiler phase is that it allows a GPU programmer to exploit wavefront splitting while obtaining low WCET. We integrate our compiler phase into the PWSG compiler flow and evaluate it on an extensive set of synthetic benchmarks. Our experiments reveal that our compiler phase can reduce the WCET of a kernel by an average of 52% compared to traditional architectures and 34% compared to the next-best selection algorithm.

References

[1] P. Bannon et al., "Computer and redundancy solution for the full self-driving computer," in *IEEE Hot Chips Symposium*, 2019, pp. 1–22.
[2] A. Klashtorny et al., "Predictable GPU wavefront splitting for safety-critical systems," *ACM TECS*, vol. 22, no. 5s, Sep 2023.
[3] ——, "PWS GPU," https://github.com/caesr-uwaterloo/pws, 2023.
[4] T. Aamodt et al., *General-Purpose Graphics Processor Architecture*. Morgan & Claypool, 2018, ch. 2, pp. 9–20.
[5] J. Meng et al., "Dynamic warp subdivision for integrated branch and memory divergence tolerance," in *Proceedings of ISCA*, 2010, pp. 235–246.
[6] M. Rhu and M. Erez, "The dual-path execution model for efficient GPU control flow," in *Proceedings of HPCA*, 2013, pp. 591–602.
[7] N. Brunie et al., "Simultaneous branch and warp interweaving for sustained GPU performance," in *Proceedings of ISCA*, 2012, pp. 49–60.
[8] S. Damani et al., "GPU subwarp interleaving," in *IEEE International Symposium on High-Performance Computer Architecture*, 2022, pp. 1184–1197.
[9] W. Fung et al., "Dynamic warp formation and scheduling for efficient GPU control flow," in *Proceedings of MICRO*, 2007, pp. 407–420.
[10] V. Hirvisalo, "On static timing analysis of GPU kernels," in *14th International Workshop on Worst-Case Execution Time Analysis*, ser. OpenAccess Series in Informatics (OASIcs), vol. 39, 2014, pp. 43–52.
[11] A. Betts et al., "Estimating the WCET of GPU-accelerated applications using hybrid analysis," in *Proceedings of ECRTS*, 2013, pp. 193–202.