Divergence Analysis and Optimizations

Bruno Coutinho, Diogo Sampaio, Fernando Magno Quintão Pereira and Wagner Meira Jr.
Department of Computer Science – Universidade Federal de Minas Gerais – Brazil
{coutinho,sampaio,fpereira,meira}@dcc.ufmg.br

***Abstract*—The growing interest in GPU programming has brought renewed attention to the Single Instruction Multiple Data (SIMD) execution model. SIMD machines give application developers a tremendous computational power; however, the model also brings restrictions. In particular, processing elements (PEs) execute in lock-step, and may lose performance due to divergences caused by conditional branches. In face of divergences, some PEs execute, while others wait; this alternation ending when they reach a synchronization point. In this paper we introduce divergence analysis, a static analysis that determines which program variables will have the same values for every PE. This analysis is useful in three different ways: it improves the translation of SIMD code to non-SIMD CPUs, it helps developers to manually improve their SIMD applications, and it also guides the compiler in the optimization of SIMD programs. We demonstrate this last point by introducing branch fusion, a new compiler optimization that identifies, via a gene sequencing algorithm, chains of similarities between divergent program paths, and weaves these paths together as much as possible. Our implementation has been accepted in the Ocelot open-source CUDA compiler, and is publicly available. We have tested it on many industrial-strength GPU benchmarks, including Rodinia and the Nvidia's SDK. Our divergence analysis has a 34% false-positive rate, compared to the results of a dynamic profiler. Our automatic optimization adds a 3% speed-up onto parallel quicksort, a heavily optimized benchmark. Our manual optimizations extend this number to over 10%.**

I. Introduction

The Flynn's taxonomy [17] defines a model of computer organization called Single Instruction, Multiple Data, or SIMD. In this paradigm we have many *processing elements* (PEs) executing in lock-step. Although SIMD programming has been the focus of much research, the hardware was expensive, and the model left the spotlight on the nineties, yet remaining important in heterogeneous, performance intensive, architectures [14], [33]. However, the increasing programmability and the low cost of graphics processing units (GPUs) are boosting once again the interest on SIMD architectures [19], [29].

SIMD machines are usually very parallel; however, the restriction that processing elements must run in lock-step makes it hard for application developers to fully exploit this processing power. A difficulty, in this case, stems from a phenomenon called *divergence*, which happens when PEs follow different paths after executing a branch instruction. Given that the hardware issues only one instruction at a time, in face of divergences, some PEs will have to wait, idly, while others execute. Divergences may be a major source of performance degradation. As an example, Baghsorkhi *et al.* [3] have analytically found that approximately one third of the execution time of the prefix scan benchmark [20], included in the CUDA software development kit (SDK), is lost due to divergences. Optimizing an application to avoid divergences is problematic for a number of reasons. First, some parallel algorithms are inherently divergent; thus, threads will naturally disagree on the outcome of branches. Second, finding highly divergent branches burdens the application developer with a tedious task, which requires a deep understanding of code that might be very complex.

The objective of this paper is to move the task of discovering divergent code away from application developers. To achieve this goal, in Section III we describe a static analysis that determines which conditional branches will never cause divergences. Our analysis works for a range of programming languages that might target SIMD hardware, from ClearSpeed's accelerators [14] to Nvidia's GPUs [29]. The GPU programming environment, for instance, groups threads into units called *warps*, which execute in lock-step. In order to delineate our model of interest, we formalize it in Section II.

Our *divergence analysis* improves the variance analysis recently introduced by Stratton *et al.* [36], which could produce false negatives. The need for these analyses emerged from compilers that try to translate CUDA programs to multi-core CPUs, such as Ocelot [13], PGI's CUDA-x86 and Stratton's version of Open64 [36]. Contrary to GPUs, CPUs do not provide hardware support for divergences, which must be handled via expensive software techniques, such as branch-on-superword-condition-codes [34]. The divergence analysis shields non-divergent branches from this overhead. Our particular motivation for this analysis, however, is different: we want to help the developer of SIMD programs to produce more efficient code. To achieve this goal, our analysis guides automatic compiler optimizations and helps developers to improve their programs via manual optimizations. To demonstrate these claims, we have implemented the techniques presented in this paper on top of Ocelot [22], a compiler that optimizes PTX - Nvidia's intermediate program representation. In Section IV we describe a few compiler optimizations that benefit from divergence analysis, and in Section IV-A we introduce *branch fusion*, an optimization that relies on the Smith-Waterman gene-sequencing algorithm [35] to identify chains of similar assembly instructions in divergent paths. The compiler can merge automatically these similar sequences into common, non-divergent, blocks of code. As we show in Section V, the divergence analysis provides a 34% false-positive rate when compared to a dynamic CUDA profiler. In terms of speed-up, branch fusion gave us gains of about 4% in well-known benchmarks, such as Cederman's parallel quicksort [8].

We have also merged divergent paths from traditional CUDA benchmarks by hand, boosting up these gains to 10%.

II. μ-SIMD: A SIMPLE SIMD LANGUAGE

In order to prove the Theorems that we state in this paper, we adopt the same model of SIMD execution independently described by Bougé *et al.* [5] and Farrell *et al.* [15]. We have a number of *processing elements* (PEs) executing instructions in lock-step, yet subject to *partial execution*. In the words of Farrel *et al.*, "All PEs execute the same statement at the same time with the internal state of each PE being either active or inactive." [15, p.40]. The archetype of a SIMD machine is the ILLIAC IV Computer [6], and there exist many old programming languages that target this model [1], [6], [7], [23], [24], [26], [32]. The recent developments in graphics cards have brought new members to this family. The *Single Instruction Multiple Threads* (SIMT) [19], [28], [29] execution model, a term made popular by Nvidia's GPUs, is currently implemented as a multi-core SIMD machine – CUDA being a programming language that coordinates many SIMD processors. We formalize the SIMD execution model via a core language that we call μ-SIMD, and whose syntax is given in Figure 1. We do not reuse the formal semantics of Bougé *et al.* or Farrell *et al.* because they assume high-level languages, whereas our techniques are better described at the assembly level. Notice that our model will not fit vectorial instruction, popularly called SIMD, such as Intel's SSE extensions, because they do not support partial execution, rather following the semantics of Carnegie Mellon's Vcode [4]. An interpreter for μ-SIMD, written in Prolog, plus many example programs, are available in our webpage [31].

(Labels – *Lbl* $\subset \mathbb{N}$)	::=	$\{l_1, l_2, \ldots,\}$
(Variables – *Var*)	::=	$\texttt{tid} \cup \{v_1, v_2, \ldots,\}$
(Instructions – *Inst*)	::=	
– (conditional jump)	|	$\texttt{branch}(v, l)$
– (unconditional jump)	|	$\texttt{jump}(l)$
– (shared memory store)	|	$\texttt{store}(v, v_x)$
– (shared memory load)	|	$\texttt{load}(v, v_x)$
– (atomic increment)	|	$\texttt{atominc}(v, v_x)$
– (binary operation)	|	$\texttt{binop}(v_1, v_2, v_3)$
– (load immediate)	|	$\texttt{const}(v, n)$
– (synchronization barrier)	|	`sync`
– (halt execution)	|	`stop`

Fig. 1. The syntax of μ-SIMD instructions. The `binop` opcode denotes usual binary operations such as addition and multiplication.

We define an abstract machine to evaluate μ-SIMD programs. The state M of this machine is determined by a tuple with five elements: $(\Theta, \Sigma, \Pi, P, pc)$, which we define in Figure 2. A processing element is a pair (t, σ), uniquely identified by the natural t, referred by the special variable `tid`. The symbol σ represents the PE's local memory, a function that maps variables to integers. The local memory is individual to each PE; however, these functions have the same domain. Thus, $v \in \sigma$ denotes a vector of variables, each of them private to a PE. PEs can communicate through a shared array Σ. We use Θ to designate the set of active PEs. A program P is a map of labels to instructions. The *program counter* (*pc*) is the label of the next instruction to be executed. The machine contains a *synchronization stack* Π. Each node of Π is a tuple $(l_{id}, \Theta_{done}, l_{next}, \Theta_{todo})$ that denotes a point where divergent PEs must synchronize. These nodes are pushed into the stack when the PEs diverge in the control flow. The label l_{id} denotes the conditional branch that caused the divergence, Θ_{done} are the PEs that have reached the synchronization point, whereas Θ_{todo} are the PEs waiting to execute. The label l_{next} indicates the instruction where Θ_{todo} will resume execution.

(Local memory)	$\sigma \subset \mathit{Var} \mapsto \mathbb{Z}$
(Shared vector)	$\Sigma \subset \mathbb{N} \mapsto \mathbb{Z}$
(Active PEs)	$\Theta \subset (\mathbb{N} \times \sigma)$
(Program)	$P \subset \mathit{Lbl} \mapsto \mathit{Inst}$
(Sync stack)	$\Pi \subset \mathit{Lbl} \times \Theta \times \mathit{Lbl} \times \Theta \times \Pi$

Fig. 2. Elements that constitute the state of a μ-SIMD program.

$$\begin{aligned}&\textbf{split}(\Theta, v) = (\Theta_0, \Theta_n) \textbf{ where}\\ &\quad \Theta_0 = \{(t,\sigma) \mid (t,\sigma) \in \Theta \textbf{ and } \sigma[v] = 0\}\\ &\quad \Theta_n = \{(t,\sigma) \mid (t,\sigma) \in \Theta \textbf{ and } \sigma[v] \neq 0\}\end{aligned}$$

$$\textbf{push}([], \Theta_n, pc, l) = [(pc, [], l, \Theta_n)]$$

$$\begin{aligned}&\textbf{push}((pc', [], l', \Theta_n') : \Pi, \Theta_n, pc, l) = \Pi' \textbf{ if } pc \neq pc'\\ &\quad \textbf{where } \Pi' = (pc, [], l, \Theta_n) : (pc', [], l', \Theta_n') : \Pi\end{aligned}$$

$$\textbf{push}((pc, [], l, \Theta_n') : \Pi, \Theta_n, pc, l) = (pc, [], l, \Theta_n \cup \Theta_n') : \Pi$$

Fig. 3. The auxiliary functions used in the definition of μ-SIMD.

The result of executing a μ-SIMD abstract machine is a pair (Θ, Σ). Figure 4 describes the big-step semantics of instructions that change the program's control flow. A program terminates if $P[pc] = \texttt{stop}$. The semantics of conditionals is more elaborate. Upon reaching $\texttt{branch}(v, l)$ we evaluate v in the local memory of each active PE. If $\sigma(v) = 0$ for every PE, then Rule BF moves the flow to the next instruction, i.e., $pc + 1$. Similarly, if $\sigma(v) \neq 0$ for every PE, then in Rule BT we jump to the instruction at $P[l]$. However, if we get distinct values for different PEs, then the branch is *divergent*. In this case, in Rule BD we execute the PEs in the "then" side of the branch, keeping the other PEs in the sync-stack to execute them later. Stack updating is performed by the **push** function in Figure 3. Even the non-divergent branch rules update the synchronization stack, so that, upon reaching a barrier, i.e, a `sync` instruction, we do not get stuck trying to pop a node. In Rule SS, if we arrive at the barrier with a group Θ_n of PEs waiting to execute, then we resume their execution at the "else" branch, keeping the previously active PEs into hold. Finally, if we reach the barrier without any PE waiting to execute, in Rule SP we synchronize the "done" PEs with the current set of active PEs, and resume execution at the next instruction after the barrier. Notice that, in order to avoid deadlocks, we must assume that a branch and its corresponding synchronization barrier determine a *single-entry-single-exit* region in the program's CFG [16, p.329].

Figure 5 shows the semantics of the rest of μ-SIMD's

$$(\textsc{Sp}) \quad \frac{P[pc] = \texttt{stop}}{(\Theta, \Sigma, \emptyset, P, pc) \to (\Theta, \Sigma)}$$

$$(\textsc{Bt}) \quad \frac{P[pc] = \texttt{branch}(v, l) \qquad \textbf{split}(\Theta, v) = (\emptyset, \Theta) \\ \textbf{push}(\Pi, \emptyset, pc, l) = \Pi' \qquad (\Theta, \Sigma, \Pi', P, l) \to (\Theta', \Sigma')}{(\Theta, \Sigma, \Pi, P, pc) \to (\Theta', \Sigma')}$$

$$(\textsc{Bf}) \quad \frac{P[pc] = \texttt{branch}(v, l) \qquad \textbf{split}(\Theta, v) = (\Theta, \emptyset) \\ \textbf{push}(\Pi, \emptyset, pc, l) = \Pi' \qquad (\Theta, \Sigma, \Pi', P, pc + 1) \to (\Theta', \Sigma')}{(\Theta, \Sigma, \Pi, P, pc) \to (\Theta', \Sigma')}$$

$$(\textsc{Bd}) \quad \frac{P[pc] = \texttt{branch}(v, l) \\ \textbf{split}(\Theta, v) = (\Theta_0, \Theta_n) \qquad \textbf{push}(\Pi, \Theta_n, pc, l) = \Pi' \\ (\Theta_0, \Sigma, \Pi', P, pc + 1) \to (\Theta', \Sigma')}{(\Theta, \Sigma, \Pi, P, pc) \to (\Theta', \Sigma')}$$

$$(\textsc{Ss}) \quad \frac{P[pc] = \texttt{sync} \qquad \Theta_n \neq \emptyset \\ (\Theta_n, \Sigma, (pc', \Theta_0, l, \emptyset) : \Pi, P, l) \to (\Theta', \Sigma')}{(\Theta, \Sigma, (pc', \emptyset, l, \Theta_n) : \Pi, P, pc) \to (\Theta', \Sigma')}$$

$$(\textsc{Sp}) \quad \frac{P[pc] = \texttt{sync} \\ (\Theta_n, \Sigma, (\_, \emptyset, \_, \Theta_0) : \Pi, P, pc + 1) \to (\Theta', \Sigma')}{(\Theta_0 \cup \Theta_n, \Sigma, \Pi, P, pc) \to (\Theta', \Sigma')}$$

$$(\textsc{Jp}) \quad \frac{P[pc] = \texttt{jump}(l) \qquad (\Theta, \Sigma, \Pi, P, l) \to (\Theta', \Sigma')}{(\Theta, \Sigma, \Pi, P, pc) \to (\Theta', \Sigma')}$$

$$(\textsc{It}) \quad \frac{P[pc] = \iota \qquad \iota \notin \{\texttt{stop}, \texttt{branch}, \texttt{sync}, \texttt{jump}\} \\ (\Theta, \Sigma, \iota) \to (\Theta', \Sigma') \qquad (\Theta', \Sigma', \Pi, pc + 1, \Theta'', \Sigma'')}{(\Theta, \Sigma, \Pi, P, pc) \to (\Theta'', \Sigma'')}$$

Fig. 4. The semantics of μ-SIMD: control flow operations. For conciseness, when two hypotheses hold we use the topmost one.

$$(\textsc{Mm}) \ \frac{\Sigma(n_x) = n}{\Sigma \vdash n_x = n} \qquad (\textsc{Mt}) \ t, \sigma \vdash \texttt{tid} = t \qquad (\textsc{Mv}) \ \frac{v \neq \texttt{tid} \quad \sigma(v) = n}{t, \sigma \vdash v = n}$$

$$(\textsc{Tl}) \quad \frac{(t, \sigma, \Sigma, \iota) \to (\sigma', \Sigma') \qquad (\Theta, \Sigma', \iota) \to (\Theta', \Sigma")}{(\{(t, \sigma)\} \cup \Theta, \Sigma, \iota) \to (\{(t, \sigma')\} \cup \Theta', \Sigma")}$$

$$(\textsc{Ct}) \quad (t, \sigma, \Sigma, \texttt{const}(v, n)) \to (\sigma \setminus [v \mapsto n], \Sigma)$$

$$(\textsc{Ld}) \quad \frac{t, \sigma \vdash v_x = n_x \qquad \Sigma \vdash n_x = n}{(t, \sigma, \Sigma, \texttt{load}(v, v_x)) \to (\sigma \setminus [v \mapsto n], \Sigma)}$$

$$(\textsc{St}) \quad \frac{t, \sigma \vdash v_x = n_x \qquad t, \sigma \vdash v = n}{(t, \sigma, \Sigma, \texttt{store}(v, v_x)) \to (\sigma, \Sigma \setminus [n_x \mapsto n])}$$

$$(\textsc{At}) \quad \frac{t, \sigma \vdash v_x = n_x \qquad \Sigma \vdash n_x = n \qquad n' = n + 1}{(t, \sigma, \Sigma, \texttt{atominc}(v, v_x)) \to (\sigma \setminus [v \mapsto n'], \Sigma \setminus [n_x \mapsto n'])}$$

$$(\textsc{Bp}) \quad \frac{t, \sigma \vdash v_2 = n_2 \qquad t, \sigma \vdash v_3 = n_3 \qquad v_1 = n_2 \otimes n_3}{(t, \sigma, \Sigma, \texttt{binop}(v_1, v_2, v_3)) \to (\sigma \setminus [v_1 \mapsto n_1], \Sigma)}$$

Fig. 5. The operational semantics of μ-SIMD: data and arithmetic operations.

instructions. A tuple $(t, \sigma, \Sigma, \iota)$ denotes the execution of an instruction ι by a PE (t, σ). All the active PEs execute the same instruction at the same time. We model this phenomenon by showing, in Rule TL, that the order in which different PEs process ι is immaterial. Thus, an instruction such as $\texttt{const}(v, n)$ causes every active PE to assign the integer n to its local variable v. The `store` instruction might lead to a data-race, i.e., two PEs trying to write on the same location in the shared vector. In this case, the result is undefined due to Rule TL. We guarantee atomic updates via $\texttt{atominc}(v, v_x)$, which reads the value at $\Sigma(\sigma(v_x))$, increments it by one, and stores it back. This result is also copied to $\sigma(v)$, as we see in Rule AT. In Rule BP we use the symbol $\otimes$ to evaluate binary operations using the semantics usually seen in arithmetics.

III. DIVERGENCE ANALYSIS

Given a μ-SIMD program P, we are interested in determining a conservative, yet non-trivial, approximation of the set of *divergent variables*. A variable v is divergent if there exist two PEs (t_1, σ_1) and (t_2, σ_2), such that $\sigma_1(v) \neq \sigma_2(v)$ at the same moment during program execution. In order to make this definition independent on the program point where the variable is used, henceforth we will be working with μ-SIMD programs in Static Single Assignment (SSA) form [12], which we will denote by μ-SIMD$_\phi$. We use the standard notation to represent the SSA's ϕ-functions, e.g.: $v = \phi(v_1, \ldots, v_n)$; however, we will not create new labels to them. That is, if the SSA conversion of program P causes us to insert a sequence of ϕ-functions before the instruction at $P[l]$, then we represent all these ϕ-functions with the label l. This arrangement indicates the fact that ϕ-functions are just an abstraction, not present in a concrete assembly program, and simplify Algorithm 7, which we will introduce soon. Using a simple case analysis plus induction on the rules given in Figure 5 we can prove that Theorem 3.1 gives the set of divergent variables.

Theorem 3.1: A variable $v \in P$ is divergent if, and only if, one of these conditions holds:

1) $v = \texttt{tid}$, see Rule MT, Figure 5.
2) v is defined atomically, e.g.: $\texttt{atominc}(v, v_x)$. See Rule AT, Figure 5.
3) v is *data dependent* on some divergent variable.
4) v is *sync dependent* on some divergent variable.

proof: See our webpage [31]. □

To explain Theorem 3.1 we must clarify the notions of *data dependence* and *sync dependence*. Given a program P, a variable $v \in P$ is data dependent on a variable $u \in P$ if P contains some assignment instruction $P[l]$ that defines v and uses u, e.g., $P[l] = \texttt{binop}(v, u, u)$. We will be using the program in Figure 6 to illustrate the concepts in this section. In this example, we see that variable i_0 is divergent, because it is data dependent on `tid`. Similarly, i is divergent, given that it is data dependent on i_0. The problem of determining the transitive closure of the set of divergent variables – given the propagation of data dependences only – is a type of program slicing [38], which can be solved by Stratton *et al.*'s [36, p.115] *Variance Analysis*. However, the variance analysis might yield false-negatives, that is, this analysis might report that a variable that shows actual divergent behavior is non-divergent. This omission happens because variance analysis misses a phenomenon that we have called sync dependences, which we introduce in Definition 3.2.

In order to define sync dependences, we need to dig out some terms well known to compiler writers. A label l_p post-dominates a label l if, and only if, every path from l to the exit of the program – i.e., the `stop` instruction – goes across l_p. Furthermore, we say that l_p is the *immediate post-dominator* of

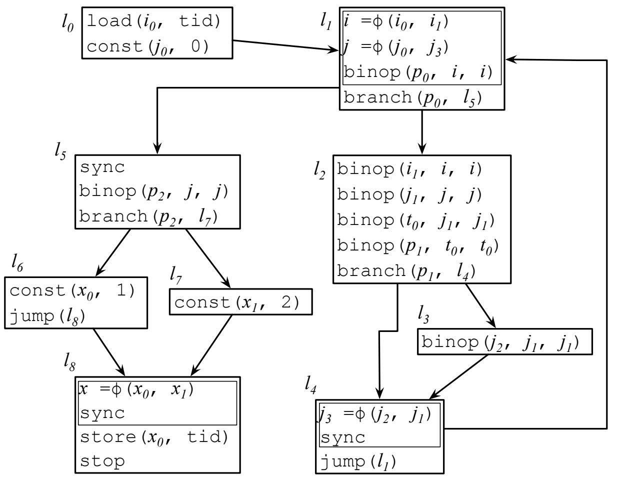


Fig. 6. A μ-SIMD$_\phi$ program. Instructions in boxes - ϕ-functions followed by a μ-SIMD instruction - correspond to one label only.

l if $l_p \neq l$, and any other label that post-dominates l also post-dominates l_p. Fung *et al.* [18] have shown that re-converging divergent PEs at the immediate post-dominator of the divergent branch is nearly optimal with respect to maximizing hardware utilization. Although Fung *et al.* have discovered situations in which it is better to do this re-convergence past l_p, they are very rare. Thus, we assume that the immediate post-dominator l_p of a divergent branch always contains a `sync` instruction.

Definition 3.2: If $P[l] = \texttt{branch}(p, l')$, and $P[l_p] = \texttt{sync}$ is the corresponding barrier, then v is *sync dependent* on p if, and only if, two different PEs may have different values for v at l_p, depending on how they branch at l.

Given a μ-SIMD$_\phi$ program P, we say that a label l *goes to* a label l', what we indicate by $l \rightarrow l'$, if one of these three conditions hold: (i) $l' = l + 1$ and $P[l] \neq \texttt{jump}(l'')$, (ii) $P[l] = \texttt{branch}(v, l')$, or (iii) $P[l] = \texttt{jump}(l')$. We say that $l_1 \overset{*}{\rightarrow} l_n$ if, either $l_1 = l_n$, or $l_1 \rightarrow l_2$ and $l_2 \overset{*}{\rightarrow} l_n$. If l is a label, and l_p is its immediate post-dominator, then we define the *influence region* $IR(l)$ as the union of every path $l \overset{*}{\rightarrow} l_p$ that contains l_p exactly once (and thus at the end). Theorem 3.3 characterizes sync divergences in terms of the structure of the control flow graph of a SSA-form program P. We use the liveness definition of Cytron *et al.* [12, p.479].

Theorem 3.3: Let P be a μ-SIMD$_\phi$ program, let $P[l] = \texttt{branch}(p, l')$, and let l_p be the immediate post-dominator of l. A variable v is sync dependent on p if, and only if, v is alive past l_p, and v is defined by a ϕ-function that reads a variable defined in $IR(p)$.

proof: See our webpage [31]. □

Notice that sync dependences are a consequence of *control dependences*, as defined by Ferrante *et al.* [16, p.323]; however, these definitions are not the same. In Figure 6, labels l_6 and l_7 are control dependent on label l_5, whereas label l_8 is not. However, variable x, defined at l_8 is sync dependent on predicate p_2, which controls $\texttt{branch}(p_2, l_7)$.

A. Computing divergent variables via graph reachability

We discover the set of divergent variables in a μ-SIMD$_\phi$ program traversing its *data dependence graph*, which we define as follows:

- for each variable $v \in P$, let n_v be a vertex of G;
- if P contains an instruction that defines variable v, and uses variable u, then we add an edge from n_u to n_v.

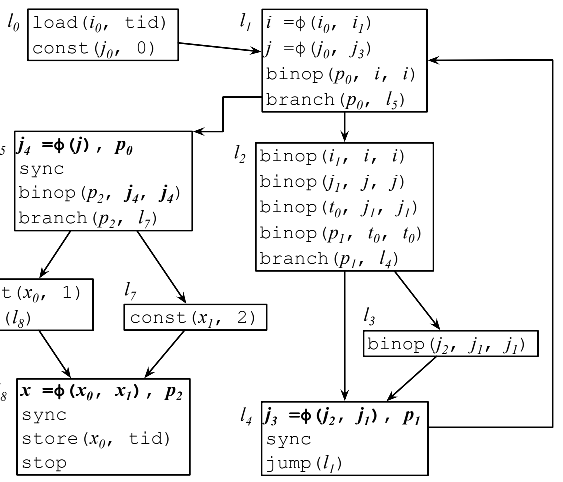
To find the divergent variables of P, we start from n_{tid}, plus the nodes that represent variables defined by atomic instructions, and mark every variable that is reachable from this set of nodes. Notice, however, that so far we are not taking sync dependences into consideration. We handle sync dependencies in our graph reachability framework by converting them into data dependences. We accomplish this conversion via an old intermediate representation called *Gated SSA form* (GSA) [30]. The GSA form defines, in addition to Cytron's ϕ-functions, three new special instructions: μ, γ and η functions, out of which only the last two have a use for us. However, to avoid the notational complexity, we will represent γ and η-functions by *gated* ϕ-functions. Thus, we augment the syntax of the standard Cytron's ϕ-function with the predicates that *control* [30, Sec 2.2] the assignment of values in that ϕ, i.e: $v = \phi(v_1, \ldots, v_n), p_1, \ldots, p_k$.

By gating ϕ-functions in this way, we add a data dependence between p_i and v; effectively reducing sync dependences to data dependences. We do not gate every ϕ-function in the program, and we may split the live ranges of some variables, via single parameter ϕ-functions, to indicate that a variable may cause divergences only in part of its live range. So, which ϕ-functions to change, and where to split live ranges, to get a program representation that allows us to solve the divergence analysis as precisely as possible? We answer this question via the algorithm in Figure 7, which we run once per branch in the program. Algorithm 7 separates, for each instruction $P[l] = \texttt{branch}(p, l)$, the set of variables that are defined inside $IR(p)$. If $P[l]$ is a forward branch, i.e., created due to the compilation of an "if-then-else" instruction, then we simply use p to gate every ϕ-function at $P[l]$'s immediate post-dominator, as we show in step (1.a). This gated ϕ-function corresponds, in Ottenstein *et al.*'s notation, to a γ-function [30, p.260], although the value of the predicate – true or false – is immaterial to us. If $P[l]$ implements a loop, then we split the live range of any variable that is used outside this loop, according to step (1.b) of our algorithm. We use single-parameter ϕ-functions to perform this live range splitting, and only the variables defined by these ϕ-functions, not necessarily their parameters, are sync dependent on p. The point where we insert this new ϕ-function corresponds to the point where Ottenstein *et al.* would insert an η-function [30, p.260]. Figure 8 shows the result of running Algorithm 7 on the program in Figure 6. The ϕ-functions at l_4 and l_8 were marked by step (1.a) of this algorithm. Variable j_3 is sync dependent on variable p_1, and variable x is sync dependent on variable p_2. The ϕ-function of arity one at l_5 was created by step (1.b.i) of Algorithm 7.

Algorithm 7: gate ϕ-functions – For each instruction $P[l] = \texttt{branch}(v, l')$ with an immediate post-dominator l_p, we do:

1) For each variable v, defined in $IR(p)$, reaching l_p do:
   a) if v is used in l_p as a parameter of a ϕ-function $v' = \phi(\ldots, v, \ldots)$, gated or not, then replace this instruction by a new ϕ-function gated by p.
   b) if v is used by an assignment instruction $x = f(\ldots, v, \ldots)$, at l_p or at some label l_x which l_p dominates, then we perform the following actions:
      i) we split the live range of v, inserting an instruction $v' = \phi(v, \ldots, v), p$ at l_p, with a parameter for each predecessor of l_p;
      ii) we rename every use of v to v' at l_p, or at any block that is dominated by l_p;
      iii) we reconvert the source program to SSA form, a necessary action due to the renaming performed in the previous step.

Fig. 7. The algorithm that transforms sync dependences into data dependences via the gating of ϕ-functions.


```
l0  load(i0, tid)
    const(j0, 0)

l1  i =ϕ(i0, i1)
    j =ϕ(j0, j3)
    binop(p0, i, i)
    branch(p0, l5)

l5  j4 =ϕ(j), p0
    sync
    binop(p2, j4, j4)
    branch(p2, l7)

l2  binop(i1, i, i)
    binop(j1, j, j)
    binop(t0, j1, j1)
    binop(p1, t0, t0)
    branch(p1, l4)

l6  const(x0, 1)
    jump(l8)

l7  const(x1, 2)

l3  binop(j2, j1, j1)

l8  x =ϕ(x0, x1), p2
    sync
    store(x0, tid)
    stop

l4  j3 =ϕ(j2, j1), p1
    sync
    jump(l1)
```

Fig. 8. The program from Figure 6, after we perform the gating of ϕ-functions. Changes are marked in boldface.

Once we have used Algorithm 7 to gate the ϕ-functions in the source program, we extract its data dependence graph. Moving on with our example, Figure 9 shows the graph created for the program in Figure 8. Surprisingly, we notice that the instruction $\texttt{branch}(p_1, l_4)$ cannot cause a divergence, even though the predicate p_1 is data dependent on variable j_1, which is created inside a divergent loop. Indeed, variable j_1 is non-divergent, although the variable p_0 that controls the loop is. We prove the non-divergence of j_1 by induction on the number of loop iterations. In the first iteration, every thread sees $j_1 = j_0 \otimes j_0$, as we infer from Rule BP in Figure 5. Assuming that at the n-th iteration every thread still in the loop sees the same value of j, then, the assignment $j_1 = j \otimes j$ concludes the induction step. Nevertheless, variable j may cause a divergence at $\texttt{branch}(p_2, l_7)$, because it is defined by

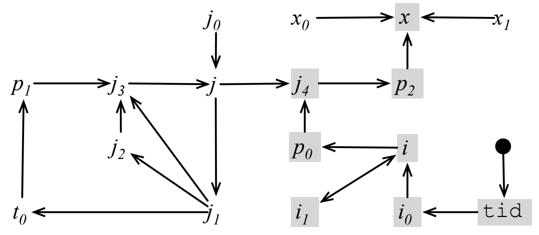


Fig. 9. The input dependence graph created for the program in Figure 8. Divergent variables are colored grey.

a ϕ-function which uses j_1. That is, once the PEs synchronize at l_5, they might have re-defined j_1 a different number of times. Although this fact cannot cause a divergence inside $IR(p_0)$, as Algorithm 7 shows, divergences might happen outside the loop controlled by p_0. Thus, we split the live range of j outside the loop via the ϕ-function at l_5 in Figure 8.

Complexity Analysis Our algorithm runs in two steps. First we change the program's intermediate representation, and then we perform a graph traversal to identify divergent variables. Changing the program involves, in the worst case, traversing its control flow graph once per each branch. If we have $|B|$ branches in a μ-SIMD program, then the CFG traversal is $O(|B|)$. For each branch we might have to rename variables and reconvert the program into SSA-form, what is $O(\alpha|V|)$ [2], where α is the inverse Ackermann function, and $|V|$ is the number of variables in the source program. Thus, Algorithm 7 runs in $O(\alpha \times |B|^2 \times |V|)$. The second phase runs in worst case $O(|V|^2)$, although we will have $O(|V|)$ on the average. Therefore, we solve the divergence analysis problem in $O(\alpha \times |B|^2 \times |V| + |V|^2)$.

IV. Divergence Optimizations and Branch Fusion

We call *divergence optimizations* the techniques that use information from a divergence analysis to improve SIMD code. Although divergence analysis is new, the literature contains examples of divergence optimizations, such as *thread reallocation*, *work unification*, and *peephole optimizations*, which we briefly discuss. In Section IV-A we introduce a novel divergence optimization, called *branch fusion*.

Thread reallocation applies on models that group PEs into independent SIMD front-lines, such as Nvidia's GPUs, which consist of a number of warps having 32 PEs that execute in lock-step. In face of divergences, one can re-group these PEs, so that only one front-line contains divergent PEs; Fung *et al.* [18] have proposed a way to perform this reallocation at the hardware level, inferring, via simulation, speed-ups of up to 20%. Zhang *et al.* [39] have implemented this reallocation by manually moving data around, before sending it to the GPU, obtaining performance gains of up to 37%.

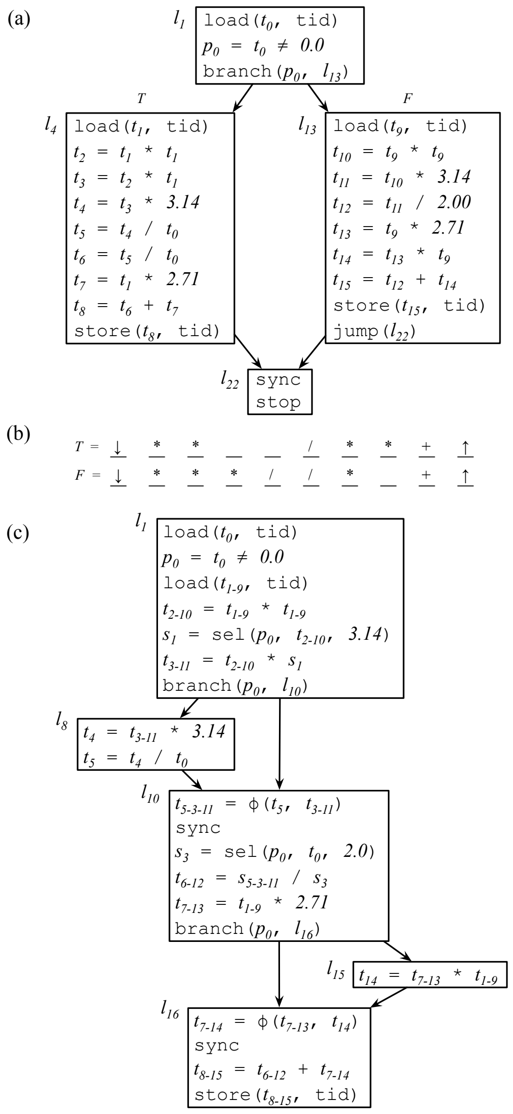
Work unification consists in leaving to only one PE the task of computing the values of non-divergent variables. Collange *et al.* [10] have designed a hardware mechanism that dynamically detects non-divergent variables and assign their computation to a single PE; hence, reducing memory accesses and hardware occupancy. This technique identifies about 19% of the values read into local registers as non-divergent.

Peephole optimizations consist in replacing instructions that are proved to be non-divergent by more efficient operations. For instance, the implementation of μ-SIMD's conditional branch instructions is complicated by the necessity to handle divergences, as one can check from the Rules BT, BP, BD, SS and SP in Figure 4. However, this complication should not be imposed on non-divergent branches. Thus, many SIMD architectures provide two conditional jumps: one that, similarly to μ-SIMD's `branch` and `sync` instructions, handles divergences, and another that does not. For instance, PTX, Nvidia's intermediate representation, provides `bra.uni`, a conditional jump that can be used in the absence of divergences. Similarly, PTX offers uniform versions of other control flow instructions, such as function calls and function returns. It also provides a uniform load (`ldu`), which assumes that every thread in the warp is reading data from the same address.

A. Branch Fusion

The purpose of branch fusion is to merge common code extracted from divergent program paths. To explain this optimization, we leave the example from Figure 6 aside to focus, instead, on the example in Figure 10. To facilitate our explanation, we replace the `binop` instruction with actual operands of obvious semantics. The sequences of instructions in the paths starting at labels l_4 and l_{13} have many operations – and a few operands – in common. Let's represent a store instruction by $\uparrow$, and a load instruction by $\downarrow$. Thus, we can represent the two sequences of instructions in the branch of Figure 10(a) by $T = \{\perp, \downarrow, *, *, *, /, /, *, +, \uparrow\}$ and $F = \{\perp, \downarrow, *, *, /, *, *, +, \uparrow\}$. These two sequences share many common subsequences, and Figure 10(b) shows a maximum matching between them. We can use this matching to transform the program in Figure 10(a) into the program in Figure 10(c). Notice that we have augmented μ-SIMD with ternary selectors (`sel`), whose purpose is to choose the operands of merged instructions, and that follow the C/C++/Java's semantics. A similar effect can be obtained in architectures that do not offer selectors, via the predicates created by *if-conversion* [21], [34]. In the rest of this section we describe a systematic way to perform the transformation that takes us from Figure 10(a) to Figure 10(c).

It is meaningful to merge only branches that force divergent PEs to be active at separate times. We call these branches "if-then-else's", to distinguish them from "if-then" and "while" branches. For instance, the instruction $\texttt{branch}(p_1, l_4)$ from Figure 6 is an "if-then" branch: had this branch been divergent, then some PEs would have to execute label l_3; however, the other PEs would not have an exclusive code path to process. In this case, no PE performs redundant work. Still in Figure 6, the possibly divergent instruction $\texttt{branch}(p_0, l_5)$ is a "while" branch. PEs that escape the loop will have to wait for the PEs that have iterations to perform. Finally, $\texttt{branch}(p_2, l_7)$ is an "if-then-else" branch. In face of a divergence some PEs will



l_1:
```
load(t0, tid)
p0 = t0 ≠ 0.0
branch(p0, l13)
```

l_4:
```
load(t1, tid)
t2 = t1 * t1
t3 = t2 * t1
t4 = t3 * 3.14
t5 = t4 / t0
t6 = t5 / t0
t7 = t1 * 2.71
t8 = t6 + t7
store(t8, tid)
```

l_{13}:
```
load(t9, tid)
t10 = t9 * t9
t11 = t10 * 3.14
t12 = t11 / 2.00
t13 = t9 * 2.71
t14 = t13 * t9
t15 = t12 + t14
store(t15, tid)
jump(l22)
```

l_{22}:
```
sync
stop
```

$T = \underline{\downarrow}\ \underline{*}\ \underline{*}\ \underline{\ \ }\ \underline{\ \ }\ \underline{/}\ \underline{*}\ \underline{*}\ \underline{+}\ \underline{\uparrow}$

$F = \underline{\downarrow}\ \underline{*}\ \underline{*}\ \underline{*}\ \underline{/}\ \underline{/}\ \underline{*}\ \underline{\ \ }\ \underline{+}\ \underline{\uparrow}$

l_1:
```
load(t0, tid)
p0 = t0 ≠ 0.0
load(t1-9, tid)
t2-10 = t1-9 * t1-9
s1 = sel(p0, t2-10, 3.14)
t3-11 = t2-10 * s1
branch(p0, l10)
```

l_8:
```
t4 = t3-11 * 3.14
t5 = t4 / t0
```

l_{10}:
```
t5-3-11 = ϕ(t5, t3-11)
sync
s3 = sel(p0, t0, 2.0)
t6-12 = s5-3-11 / s3
t7-13 = t1-9 * 2.71
branch(p0, l16)
```

l_{15}:
```
t14 = t7-13 * t1-9
```

l_{16}:
```
t7-14 = ϕ(t7-13, t14)
sync
t8-15 = t6-12 + t7-14
store(t8-15, tid)
```

Fig. 10. (a) The example that we use to explain branch fusion. (b) A possible instruction alignment. (c) Code after branch fusion.

execute label l_6, while others will execute label l_7. In this case, each group of divergent PEs goes over an exclusive code path. We call the "if-then-else" branches *unifiable*, and we recognize them in the following way:

Definition 4.1: UNIFIABLE BRANCH If $l = \texttt{branch}(v, l_1)$ is a conditional jump with two successor labels l_1 and l_2, then this branch is unifiable if, and only if, there exists no path $l_1 \xrightarrow{*} l_2 \in IR(l)$, and there exists no path $l_2 \xrightarrow{*} l_1 \in IR(l)$.

The effectiveness of branch fusion depends on the solution of the *Instruction Alignment Problem*, a variation of sequence alignment [27] that we define as follows:

Definition 4.2: 2D INSTRUCTION ALIGNMENT

Instance: given the following input parameters:

- two arrays of instructions, $T = \{i_1, \ldots, i_n\}$ and $F = \{j_1, \ldots, j_m\}$;

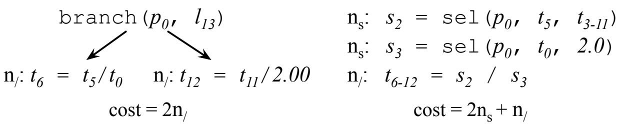

branch(p_0, l_{13})
$n_/$: $t_6 = t_5/t_0$
$n_/$: $t_{12} = t_{11}/2.00$
cost = $2n_/$
n_s: s_2 = sel(p_0, t_5, t_{3-11})
n_s: s_3 = sel(p_0, t_0, 2.0)
$n_/$: $t_{6-12} = s_2 / s_3$
cost = $2n_s + n_/$

Fig. 11. Comparing the costs of divergence and merging.

- a 2-ary scoring function s, such that $s(i, j)$ is the profitability of merging instructions i and j;
- a 2-ary cost function c, such that $c(i, j)$ is the cost of the selectors necessary to merge i and j;
- b, the cost of an *alignment gap*, which, in our case, is the cost of inserting a branch into the code.

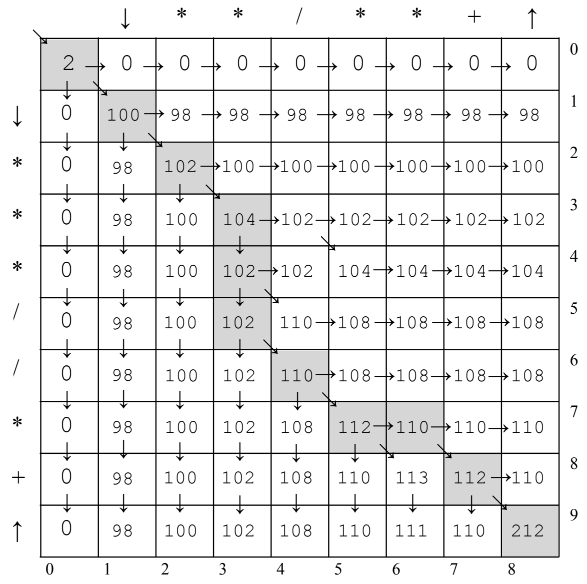
Problem: find an ordered sequence of pairs $A = \langle(x_1, y_1), ..., (x_k, y_k)\rangle$, such that:

- if $(x, y) \in A$, then $1 \leq x \leq n, 1 \leq y \leq m$
- if $r > s$ then $x_r \geq x_s$
- if $r > s$ then $y_r \geq y_s$
- $\sum(s(x, y) - c(x, y)) - b \times G$ is maximum, where $(x, y) \in A$, and G is the number of gaps in the alignment.

We solve the bi-dimensional instruction alignment problem via the classic Smith-Waterman's [35] algorithm for sequence alignment. This algorithm has two steps: first, we build a *profitability matrix* that assigns gains to each possible pairing of instructions. Next, we traverse this matrix, backwardly, in order to discover the best overall instruction alignment.

a) Computing the Profitability Matrix: There is a gain and a cost, measured in terms of processor cycles, involved in the extraction of a pair of instructions from a divergent path. Figure 11 illustrates the operations necessary to merge two division operations located in different paths of Figure 10(a). On the left we have the original branch, and on the right the program after branch fusion. The gain of merging two instructions, which we represent by a *scoring function* s is the number of cycles saved by not having to execute these instructions separately. In our example, we are saving one division; hence $s(/, /) = n_/$. The cost c is the number of cycles taken by a selector, times the number of selectors necessary to merge the instructions. We need two selectors; hence, we have $c(t_6 = t_5/t_0, t_{12} = t_{11}/2.0) = 2n_s$. Normally we derive these numbers from the programmers manual. We say that the fusion is *profitable* if the cost of executing the merged instruction is less than the cost of executing the divergent code. The profit in Figure 11 is $n_/ - 2n_s$.

In order to fuse divergent instructions, we must find the most profitable sequence of instructions that can be merged. Our guide in this search is the *profitability matrix*. To produce the profitability matrix, we write one instruction sequence along the top row, and the other sequence along the leftmost column of a bi-dimensional matrix. Each cell of the matrix is associated with a value g, e.g.: $H[i, j] = g$, where g is the maximum profit of any possible way to merge instructions up to the indices i and j. We compute $H[i, j]$ via the following recurrence relation, where the meaning of the parameters s, c and b is explained in Definition 4.2:



		↓	*	*	/	*	*	+	↑	
	2	0	0	0	0	0	0	0	0	0
↓	0	100	98	98	98	98	98	98	98	1
*	0	98	102	100	100	100	100	100	100	2
*	0	98	100	104	102	102	102	102	102	3
*	0	98	100	102	102	104	104	104	104	4
/	0	98	100	102	110	108	108	108	108	5
/	0	98	100	102	110	108	108	108	108	6
*	0	98	100	102	108	112	110	110	110	7
+	0	98	100	102	108	110	113	112	110	8
↑	0	98	100	102	108	110	111	110	212	9
	0	1	2	3	4	5	6	7	8	

Fig. 12. The profitability matrix for program in Figure 10(a).

$$H[i, j] = s(i, j) + \text{MAX}\begin{cases} H[i-1, j-1] - c(i, j) - b, \\ H[i, j-1], \\ H[i-1, j], \\ 0 \end{cases}$$

Figure 12 shows the profitability matrix to the program in Figure 10(a). We are using the following scoring function: $s(\downarrow, \downarrow) = s(\uparrow, \uparrow) = 100$, $s(*, *) = 2$, $s(/, /) = 8$, $s(+, +) = 2$. We assume that the cost of inserting a branch is $b = 2$. Notice that this cost is only paid once, when we leave a sequence of diagonal cells and go to either a vertical or horizontal cell.

Once we have computed the profitability matrix we find a solution to the instruction alignment problem in two steps. First, we scan the matrix to find the cell $H[x, y]$ with the highest profit. Second, we traverse the matrix, starting from $H[x, y]$ and going backwards towards $H[0, 0]$, following the direction of updates. That is, if $H[i, j]$ has been updated from $H[i - 1, j - 1]$, then we continue our traversal from the latter cell. In our example, the most profitable cell is $H[9, 8]$, and the most profitable sequence is $A = \langle(1, 1), (2, 2), (3, 3), (4, 3), (5, 4), (6, 4), (7, 5), (7, 6), (8, 7), (9, 8)\rangle$. We have augmented each cell of Figure 12 with the direction of its update. The gray boxes mark the most profitable path in the matrix, which denotes exactly the alignment seen in Figure 10.

We are allowed to merge instructions with different opcodes, as long as there exists an identity between these operands. For instance, the comparison operator: $p = a > b$ is equivalent to $p = b < a$. As another example, in Figure 12, we let 1 be the profit of merging an addition plus a multiplication. In this case we assume the existence of a CUDA-like "multiply-add" instruction: $t = a \times b + c$, which we can use as an identity for both instructions. In our work we use only simple identities; however, it is possible to take this approach to the extreme, employing equality saturation [37] to find the most similar code sequences for each branching path.

b) Algorithmic Complexity: Given two sequences of instructions, T and F, computing the profitability matrix is $O(|T| \times |F|)$ in terms of time and space. Finding the most profitable cell has the same complexity. Walking from the most profitable cell back to the origin of the matrix is $O(|T|+|F|)$.

c) Code Generation: Given two instruction sequences $T = \{i_1, \ldots, i_n\}$ and $F = \{j_1, \ldots, j_m\}$ that follow a conditional jump $\texttt{branch}(p, l')$, plus an instruction alignment $A = \{(x_1, y_1), ..., (x_k, y_k)\}$, we use the following pattern matching rules to generate code:

- $A = \{\ldots, (x-1, y-1), (x, y), \ldots\}$: we merge instructions $T[x]$ and $F[y]$ into a new instruction at the end of the merged block of instructions;
- $A = \{\ldots, (x-1, y-1), (x, y), (x, y+1), \ldots, (x, y+k), \ldots\}$: we create a sequence of labels $l_1, \ldots, l_k$ for the instructions $F[y+1], \ldots, F[y+k]$, and add $\texttt{branch}(p, l)$ at the end of the merged block of instructions, where l is the label of the next merged block;
- $A = \{\ldots, (x-1, y-1), (x, y), (x+1, y), \ldots, (x+k, y), \ldots\}$: we create a sequence of labels $l_1, \ldots, l_k$ for the instructions $T[x+1], \ldots, T[x+k]$, and add $\texttt{branch}(p, l)$ at the end of the merged block.

Figure 10(c) shows the code that we produce after the profitability matrix given in Figure 12.

V. EXPERIMENTS

We have implemented our analysis on top of the Ocelot PTX compiler [22], version 1.0.432. PTX, the *parallel thread execution* language, is Nvidia's high-level instruction set used to represent CUDA kernels. Ocelot is an open-source tool, designed and implemented at Georgia Tech, that parses and optimizes PTX. In order to guarantee reproducibility, we have made our patches to Ocelot, and all the scripts used in these experiments available in our webpage [31]. We run these experiments on a Nvidia GeForce GTX 260 video card.

The Benchmarks: We tried to gather a meaningful collection of benchmarks that (i) are publicly available, (ii) have been used in previous works [8], [9], [22] and (iii) have been designed and implemented by experts in the area. We chose the following benchmarks: Rodinia [9], Nvidia's SDK and the parallel quicksort [8]. These benchmarks contain 30 applications, which provide over 80 kernels. A kernel is the Nvidia's jargon to a procedure that runs on the GPU. We found divergences, via dynamic profiling, in 67 kernels, which we will use in this section. These programs, which we produce using Nvidia's `nvcc` 3.0, contain together over 46,000 PTX assembly instructions.

The precision of divergence analysis: We have compared the output of the divergence analysis to the results of an instrumentation-based CUDA profiler that we have implemented [11]. The profiler runs a program plus a sample input

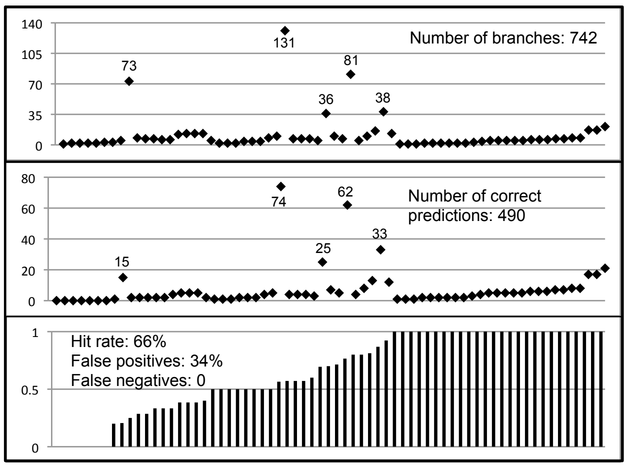


Fig. 13. Precision. (Top) number of branches per application. (Middle) number of true positives and true negatives. (Bottom) percentage of true inferences. Each dot represents a kernel. Kernels are sorted by hit-rate.

and verifies which branches have caused actual divergences during the program execution. The results of this comparison are given in Figure 13. We notice that we had a false-positive rate of 34%. However, we point out that we have exercised each of the Nvidia's SDK benchmarks with only one input data, obtaining a false-positive rate of 39%. We have used two different types of input data on Rodinia's applications, obtaining the smaller false-positive rate of 31%. A particular program input might not exercise every program path, and so we speculate that the results of divergent analysis will be closer to the results of the profiler as we test the subject program with a greater quantity of input samples. Another source of imprecision is the fact that most of the input samples that come with the benchmarks are tuned to the GPU's warp size, e.g., the input size is a multiple of 32. In this scenario, we never find divergences in code that tests specifically for situations in which the input size does not exactly fit the warp size.

The number of divergent variables: Figure 14 shows the number of divergent variables that we have found. Our 67 subject kernels, when converted into SSA form, give us a total of 38,150 variables. This number is considerably bigger than the number of variables in the original program, because the SSA representation renames each redefinition of the same variable name. The divergent analysis has found that 14,861 variables are non-divergent. This number means, for instance, that we could move 39% of all the program variables into shared memory, possibly making room for more processing elements in the graphics unit. We have found that 26% of the branches are non-divergent, and we have been able to replace these instructions by uniform instructions. A remarkable outlier in this chart is SDK's `concurrentKernels::mykernel`. Our analysis showed that only 4 variables in the kernel, out of 839, can be divergent.

Analysis run time: The divergence analysis has a worst case performance that is quadratic on the number of variables in the

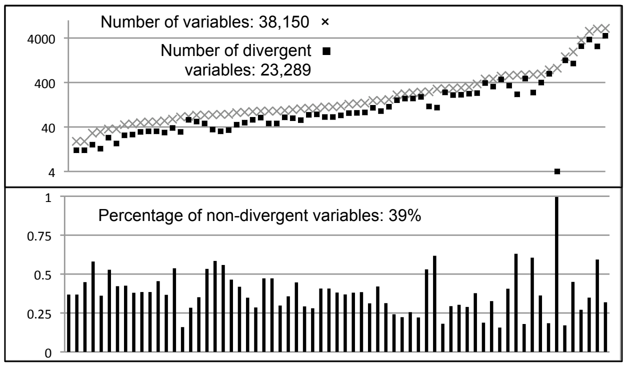


Fig. 14. (Top) Number of divergent variables (Theorem 3.1). (Bottom) Percentage of non-divergent variables. Each dot represents a kernel. Kernels are sorted by number of variables.

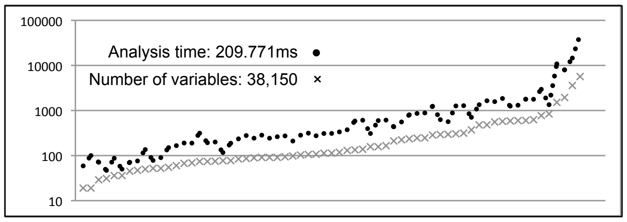


Fig. 15. Runtime(ms) of our analysis, compared to the number of program variables. Each dot is a kernel. Kernels are sorted by number of variables.

source program, because the input dependence graph contains one vertex for each variable, and it might be dense. However, in practice the size of this graph is linear in the number of variables in the source program, and our experiments show this fact. Figure 15 compares the running time of our analysis with the number of variables in the source program. We see that there is a linear correlation between the number of variables and the running time of our analysis.

The opportunities for branch fusion: The three charts in Figure 16 show static numbers that we measured after applying branch fusion onto our benchmarks. We found 196 unifiable branches in the test suite; hence, about 26% of the branches are of the "if-then-else" variety from Definition 4.1. Regarding the proportion of unifiable branches, this benchmark collection does not differ from traditional C programs. For instance, `403.gcc`, the largest SPEC CPU 2006 program, contains 2,152,106 branches, out of which 23.5% are unifiable. We got this number by traversing the bytecodes that LLVM 2.8 [25] produces after compiling `403.gcc`. The divergence analysis indicates that 70% of the unifiable branches, i.e., 137 conditional jumps, are divergent. After building the profitability matrix for all these branches, we found out 30 profitable unifications. In other words, the algorithm from Section IV-A points out that 4% of the 742 branches could be unified.

Performance gains of branch fusion: We chose the six branches, out of 30, whose unification profit exceeds 100 cycles. These branches are in `rodinia :lud` (0.37%, 1.35%),

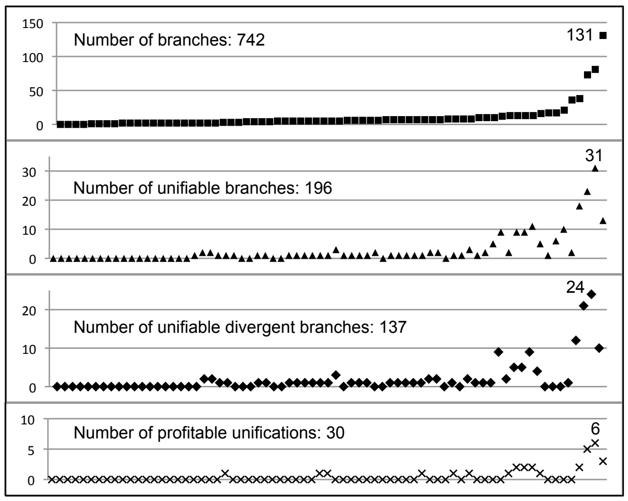


Fig. 16. Static numbers for branch fusion. Each dot is a kernel. Kernels are sorted by the number of branches.

`sdk :mergeSort` (3.59%), `rodinia :heartwall` (0.62%), `rodinia:srad` (0.48%) and parallel quicksort (3.09%). The numbers in parentheses are the speed-ups after branch fusion. Our gains are small for two reasons. First, the benchmarks are well known applications, already heavily optimized. If we apply branch fusion on naive kernels, the gains are much better. For instance, we obtain 26.78% improvement on a CUDA version of the program in Figure 10 [31]. Second, we stop unification at the basic block boundaries and do not reorder instructions. Once we start breaking these rules, as we have done by hand, and show next, the benefits are considerably superior. Nevertheless, the gains obtained automatically are larger than the speed-ups given by classic optimizations such as constant propagation.

Using divergence analysis as a developer's guiding tool: In addition to pointing to the compiler the optimization hotspots, the divergence analysis – and in particular the profitability matrix – indicate to developers where to look for code improvements. As an example, we have used it to hand optimize Cederman's implementation of quicksort [8]. This CUDA program uses the core algorithm seen in Figure 17 (a), which is visited 28 million times by the benchmark's reference data set. The branch condition is clearly divergent, as it depends on `tid`. By merging the divergent code we obtain the kernel in Figure 17 (b), which accounts for a speed-up of 9.2% on the whole quicksort implementation. That is, we deliver almost 10% speed-up by changing 12 assembly instructions in a program containing 895 instructions! Our current implementation of branch fusion does not obtain this gain, because it involves merging a path of consecutive basic blocks. We have also obtained 11.5% improvement on Rodinia's SRAD by merging the contents of different branches. The new code has been reported to the maintainer.

```
if ((tid & k) == 0) {
  if (sh[tid] > sh[ixj]) {
    swap(sh[tid], sh[ixj]);
  }
} else {
  if (sh[tid] < sh[ixj]) {
    swap(sh[tid], sh[ixj]);
  }
}
```
(a)

```
int p = (tid & k) == 0;

unsigned b = p?tid:ixj;
unsigned a = p?ixj:tid;

if (sh[b] > sh[a]) {
  swap(sh[b], sh[a]);
}
```
(b)

Fig. 17. (a) Part of the original bitonic kernel. (b) Hand optimization.

VI. Conclusion

We have presented analyses and optimizations that mitigate the negative impact of divergences in languages that fit the SIMD execution model. The development of general purpose applications in SIMD hardware is becoming increasingly popular, and we believe that our contributions will be one more ally helping developers to take maximum benefit from these powerful machines. We are currently working on the development of new divergence optimizations, such as variable sharing. As future work, we intend to increase the precision of our analysis by coupling it with Collange *et al.*'s affine analysis [10]. The implementation of divergence analysis is publicly available in the Ocelot's open source distribution. Further material about our research can be found in our webpage at `http://divmap.wordpress.com/`.

Acknowledgment: we thank Sylvain Collange and Marcelo D'Amorim for reviewing a draft of this paper, and Jacques Cohen for pointing the Smith-Waterman algorithm to us. This work was supported by CNPq, CAPES, FAPEMIG and InWeb.

References

[1] Norma E. Abel, Paul P. Budnik, David J. Kuck, Yoichi Muraoka, Robert S. Northcote, and Robert B. Wilhelmson. TRANQUIL: a language for an array processing computer. In *AFIPS*, pages 57–73. ACM, 1969.

[2] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, 2nd edition, 2002.

[3] Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wen-mei W. Hwu. An adaptive performance modeling tool for GPU architectures. In *PPoPP*, pages 105–114. ACM, 2010.

[4] Guy Blelloch and Siddhartha Chatterjee. Vcode: A data-parallel intermediate language. In *FMPC*, pages 471–480. ACM, 1990.

[5] Luc Bougé and Jean-Luc Levaire. Control structures for data-parallel SIMD languages: semantics and implementation. *Future Generation Computer Systems*, 8(4):363–378, 1992.

[6] W.J. Bouknight, Stewart A. Denenberg, David E. McIntyre, J. M. Randall, Amed H. Sameh, and Daniel L. Slotnick. The Illiac IV system. *Proceedings of the IEEE*, 60(4):369–388, 1972.

[7] Klaus Brockmann and Rolf Wanka. Efficient oblivious parallel sorting on the MasPar MP-1. *ICSS*, 1:200, 1997.

[8] Daniel Cederman and Philippas Tsigas. GPU-quicksort: A practical quicksort algorithm for graphics processors. *Journal of Experimental Algorithmics*, 14(1):4–24, 2009.

[9] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, pages 44–54. IEEE, 2009.

[10] Sylvain Collange, David Defour, and Yao Zhang. Dynamic detection of uniform and affine vectors in GPGPU computations. In *HPPC*, pages 46–55. Springer, 2009.

[11] Bruno Coutinho, Diogo Sampaio, Fernando Magno Quintao Pereira, and Wagner Meira Jr. Performance debugging of GPGPU applications with the divergence map. In *SBAC-PAD*, pages 33–40. IEEE, 2010.

[12] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490, 1991.

[13] Gregory Diamos, Andrew Kerr, Sudhakar Yalamanchili, and Nathan Clark. Ocelot, a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *PACT*, pages 354–364, 2010.

[14] T Endo and S Matsuoka. Massive supercomputing coping with heterogeneity of modern accelerators. In *IPDPS*, pages 1–10. IEEE, 2008.

[15] Craig A. Farrell and Dorota H. Kieronska. Formal specification of parallel SIMD execution. *Theo. Comp. Science*, 169(1):39–65, 1996.

[16] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *TOPLAS*, 9(3):319–349, 1987.

[17] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, C-21:948+, 1972.

[18] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *MICRO*, pages 407–420. IEEE, 2007.

[19] Michael Garland and David B. Kirk. Understanding throughput-oriented architectures. *Commun. ACM*, 53:58–66, 2010.

[20] Mark Harris. The parallel prefix sum (scan) with CUDA. Technical Report Initial release on February 14, 2007, NVIDIA, 2008.

[21] Ken Kennedy and Kathryn S. McKinley. Loop distribution with arbitrary control flow. In *Supercomputing*, pages 407–416. IEEE, 1990.

[22] Andrew Kerr, Gregory F. Diamos, and Sudhakar Yalamanchili. A characterization and analysis of PTX kernels. In *IISWC*, pages 3–12. IEEE, 2009.

[23] R. Keryell, Ph. Materat, and N. Paris. POMP, or how to design a massively parallel machine with small developments. In *PARLE*, pages 83–100. Springer, 1991.

[24] Sun-Yuan Kung, K. S. Arun, R. J. Gal-Ezer, and D. V. Bhaskar Rao. Wavefront array processor: Language, architecture, and applications. *IEEE Trans. Comput.*, 31:1054–1066, 1982.

[25] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE, 2004.

[26] Duncan H. Lawrie, T. Layman, D. Baer, and J. M. Randal. Glypnir-a programming language for Illiac IV. *Commun. ACM*, 18(3):157–164, 1975.

[27] David W. Mount. *Bioinformatics: Sequence and Genome Analysis*. Cold Sprint Harbor Laboratory Press, 1st edition, 2004.

[28] John Nickolls and William J. Dally. The GPU computing era. *IEEE Micro*, 30:56–69, 2010.

[29] John Nickolls and David Kirk. *Graphics and Computing GPUs. Computer Organization and Design, (Patterson and Hennessy)*, chapter A, pages A.1 – A.77. Elsevier, 4th edition, 2009.

[30] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. MacCabe. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *PLDI*, pages 257–271. ACM, 1990.

[31] Fernando M. Q. Pereira, 2011. `http://divmap.wordpress.com/`.

[32] R. H. Perrot. A language for array and vector processors. *TOPLAS*, 1:177–195, 1979.

[33] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

[34] Jaewook Shin. Introducing control flow into vectorized code. In *PACT*, pages 280–291. IEEE, 2007.

[35] Temple F. Smith and Michael S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195 – 197, 1981.

[36] John A. Stratton, Vinod Grover, Jaydeep Marathe, Bastiaan Aarts, Mike Murphy, Ziang Hu, and Wen-mei W. Hwu. Efficient compilation of fine-grained SPMD-threaded programs for multicore CPUs. In *CGO*, pages 111–119. IEEE, 2010.

[37] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In *POPL*, pages 264–276. ACM, 2009.

[38] Mark Weiser. Program slicing. In *ICSE*, pages 439–449. IEEE, 1981.

[39] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. On-the-fly elimination of dynamic irregularities for GPU computing. In *ASPLOS*, pages 369–380. ACM, 2011.