Function Call Re-Vectorization

Rubens E A Moreira, Caroline Collange, Fernando Magno Quintão Pereira

Function Call Re-Vectorization

Rubens Moreira, Caroline Collange, Fernando Pereira

Function Call Re-Vectorization

Rubens E. A. Moreira
UFMG – Brazil
rubens@dcc.ufmg.br

Caroline Collange
INRIA – France
caroline.collange@inria.fr

Fernando Magno Quintão Pereira
UFMG – Brazil
fernando@dcc.ufmg.br

Abstract

Programming languages such as C for CUDA, OpenCL or ISPC have contributed to increase the programmability of SIMD accelerators and graphics processing units. However, these languages still lack the flexibility offered by low-level SIMD programming on explicit vectors. To close this expressiveness gap while preserving performance, this paper introduces the notion of Call Re-Vectorization (CREV). CREV allows changing the dimension of vectorization during the execution of a kernel, exposing it as a nested parallel kernel call. CREV affords programmability close to dynamic parallelism, a feature that allows the invocation of kernels from inside kernels, but at much lower cost. In this paper, we present a formal semantics of CREV, and an implementation of it on the ISPC compiler. We have used CREV to implement some classic algorithms, including string matching, depth first search and Bellman-Ford, with minimum effort. These algorithms, once compiled by ISPC to Intel-based vector instructions, are as fast as state-of-the-art implementations, yet much simpler. Thus, CREV gives developers the elegance of dynamic programming, and the performance of explicit SIMD programming.

Categories and Subject Descriptors D - Software [*D.3 Programming Languages*]: D.3.4 Processors - Compilers

General Terms Languages, Performance

Keywords SIMD, SIMT, Function, Programmability

1. Introduction

New hardware asks for new programming idioms. As an example, the appearance of general purpose Graphics Processing Units (GPUs) has led to a revolution in programming languages [12, 23], which has culminated in the materialization of languages such as C for CUDA [29], OpenCL [22], ISPC [25] and PyCuda [18]. These Multi-Threaded (MT) languages let programmers express computations as single kernels executed by many threads. They target architectures that combine SIMD and multi-threaded execution, like GPUs and multi-core CPUs with vector instructions.

However, such high-level abstractions come at a cost in flexibility and composability. Most programming languages that target hardware accelerators pack threads into SIMD vectors or GPU warps[1] for the whole duration of a kernel call. They also suspend and resume individual threads to simulate thread-dependent control flow. These constraints affect device-side library functions, which cannot assume any particular organization of parallelism nor thread activity. For instance, invoking full SIMD functions within divergent regions might lead to incorrect behavior. Consequently, library functions often contain two versions of each routine, and dynamically dispatch the proper version depending on whether threads diverge at the call site or not[2].

On GPUs, developers circumvent the restrictions listed above in two ways: via *warp-synchronous programming*, or via *dynamic parallelism*. In the first case, programmers use the knowledge that threads are grouped in warps to achieve thread communication without synchronization or memory sharing. Yet, warp-synchronous programming is not easily composable with classic multi-thread programming. Programmers must ensure that every thread in a warp participates in each collective operation; e.g., the CUDA `__shfl` function has undefined behavior when reading data from an inactive thread. However, multi-thread programming puts thread divergences out of the hands of programmers. Consequently, such warp-synchronous functions may not be called from multi-thread code that may have divergent control flow.

CUDA's dynamic parallelism (or OpenCL's device-side enqueue) lets threads already in flight create new groups of threads [36]. This feature gives developers the opportunity to implement strikingly elegant algorithms [21]. However, this construct is too heavyweight for our simpler purpose of reactivating threads within a warp. For instance, invoking new threads from within a thread in CUDA involves the global scheduling of a new grid of threads [16], a very expensive event. In short, currently, either we have the programmability and elegance of the multi-threaded model, or the efficiency of warp-synchronous programming, but not both.

The goal of this paper is to allow the composability of SIMD and MT through a programming construct syntactically similar to dynamic parallelism. To this end, we intro-

[1] We shall call groups of threads that execute in lock-step a *warp*.

[2] Ex.: see Trove at https://github.com/bryancatanzaro/trove.

duce the notion of *Call Re-Vectorization* (CREV). CREV is a programming idiom that modifies function calls. Functions marked with the **crev** tag, henceforth called *r-functions*, are executed by all the threads in a SIMD unit. This implies a context switch: to run a r-function, the runtime must change the state of all the threads, including those inactive due to divergent control flows. Upon completion, workers return to their previous state, in the same way that a function call is handled. Thus, we achieve a new level of recursion, in which threads can spawn new threads in a stack-based fashion. However, contrary to traditional dynamic parallelism, CREV uses only the accelerator's local memory (registers and call stack) to save thread states; hence, it is cheaper.

To validate our ideas, we have implemented them in ISPC [3, 25][3]. ISPC is a programming language, plus its compiler. This compiler produces industrial quality code for SIMD units such as Intel Streaming SIMD Extensions (SSE), Intel Advanced Vector Extensions (AVX) including AVX-512 for Xeon Phi accelerators [31], or ARM NEON. We chose to implement CREV in ISPC because this framework supports the notion of *unmasked* or *everywhere* blocks [25]: the ability to activate – in a new context – threads that are idle due to divergences. This feature is a requirement of CREV. We have used this new ISPC's extension to implement several classic algorithms using CREV. We show that these implementations are as efficient as warp-synchronous versions of them, and as clear and elegant as if they had been implemented using dynamic parallelism.

Summary of our Contributions. The key contribution of this paper is the notion of Call Re-Vectorization, which comes out of the observation that it is possible to capitalize on divergent threads to help speedup the work of active threads. We explain the concept of CREV through examples, a formal semantics and an industrial quality implementation:

- **Examples:** Section 2 shows examples of algorithms that benefit from our notion of Call Re-Vectorization. Further examples are discussed in Section 4.
- **Semantics:** Section 3.2 formalizes the semantics of μ-SIMD, a low-level instruction set sufficient to implement CREV. We have written a Prolog interpreter to validate that semantics. This interpreter made it easy to prototype different implementations of CREV, until we had a design that we could graft into a state-of-the-art compiler.
- **Translation:** Section 3.3 describes the translation of the high-level "**crev**" keyword into the low-level representation. Core properties of the final, low-level code, as produced by the translator, are listed in Section 3.4.
- **Evaluation:** Section 4 provides an empirical evaluation of our implementation. To perform this evaluation, we have implemented some algorithms, which are faster and cleaner than their original versions without CREV.

[3] The Intel SPMD Program Compiler (ISPC) is available at https://github.com/ispc/

2. Overview

The goal of this section is to explain *Warp-Synchronous Programming*, *Dynamic Parallelism* (DP), and our notion of *Call Re-Vectorization* (CREV). To this end, we shall use Algorithm 1 as an example. This program receives a book b_i, plus a pattern p. It then copies out all the lines $l \in b_i$ that match p. Pattern matching is performed by `memcmp`, and memory copying is done by `memcpy`. The book is represented as a matrix of characters; thus, each of its lines, and also the pattern p, is a vector of up to N characters. Algorithm 1 runs in parallel: t_{id} is a thread identifier. Hence, each thread is in charge of matching a line l in b_i against p. In case the match is positive, this thread must copy l to an output matrix b_o. For clarity, we assume a single warp in this example, although the techniques we described can be applied independently to multiple warps. The number of threads that run simultaneously in Algorithm 1 is W, the warp width.

Algorithm 1: SIMD Book Filter

```
1 W ← warp size; t_id ← thread index;
2 Function bFilter(mtx b_i, mtx b_o, vec p, int N)
3     for k ← t_id to num_lines(b_i) - 1 step W do
4         l ← b_i[k];
5         if memcmp(l, p, N) == 0 then
6             memcpy(l, b_o[k], N);
```

A naive multi-thread implementation of `memcpy` iterates sequentially over the arrays within each thread. This implementation is highly inefficient due to branch and memory divergence. Branch divergence occurs if the number of iterations N differs across threads. Threads with few iteration would finish the loop earlier and wait for threads with more iterations in order to restore convergence at the end of the loop. Memory divergence also happens as threads within a warp access data in unrelated locations. Such accesses, referred to as *uncoalesced* in the CUDA literature or as *gather/scatter* on SIMD platforms, are bandwidth-inefficient compared to accesses to consecutive elements.

2.1 Warp Synchronous Programming

It is possible to write function `memcpy` in a way that distributes operations on contiguous elements across consecutive threads. Algorithm 2 does it. Function `memcpy_shfl` is aware of the SIMD nature of a warp. Variables are stored as vectors, having each position belonging to a specific thread. Instruction `shfl`(v, i) allows thread t_{id} to read the value stored in variable v, but in the register space of thread i. This implementation give us an efficient way to copy data between arrays, as copies are distributed evenly between threads, removing most of the branch divergence. Memory divergence is also eliminated as threads of a warp access consecutive elements at each iteration of the loop on line 7.

Nevertheless, this function has an important limitation: it requires all threads in the warp to be active. It cannot safely be called from a point that has potential branch divergence. Indeed, the loop on line 7 would skip elements if some threads were inactive. To support calls to `memcpy_shfl` within divergent regions, we need a way to re-activate threads and put them to work on the copy loop. In addition, the warp-synchronous programming construct is more complex and error-prone than the naive implementation.

Algorithm 2: Warp synchronous `memcpy`

```
1 W ← warp size; t_id ← thread index;
2 Function memcpy_shfl(vec s, vec d, int N)
3     for j ← 0 to W − 1 do
4         d_my ← shfl(d, j);
5         s_my ← shfl(s, j);
6         N_my ← shfl(N, j);
7         for i ← t_id to N_my − 1 step W do
8             d_my[i] ← s_my[i];
```

2.2 Dynamic Parallelism in CUDA

In NVIDIA's CUDA and OpenCL 2.0, dynamic parallelism (DP) is the ability to invoke a new kernel K_2 from within a kernel K_1 [32]. In this case, programmers may request a large number of threads, i.e., multiple new warps in multiple thread blocks. As the inner K_2 is a new kernel, all its threads are active upon entry, regardless of branch divergence in K_1. Algorithm 3 shows an implementation of `memcpy` that we could invoke from Algorithm 1 using dynamic parallelism. This algorithm splits, among all the threads in a warp, the work of copying vector s to vector d. Its main advantage is simplicity; its disadvantage is efficiency.

Algorithm 3: Implementation of `memcpy` that could be invoked dynamically from Algorithm 1.

```
1 W ← warp size; t_id ← thread index;
2 Function memcpy_dp(vec s, vec d, int N)
3     for k ← t_id to N − 1 step W do
4         d[k] ← s[k];
```

Wang *et al.* demonstrate that the overhead of a new kernel launch can be as high as one millisecond [32]. The new kernel must be scheduled and wait until there are resources available for its execution. Then, the requested number of warps and memory blocks must be allocated before execution starts. For large workloads, the overhead of launching a nested kernel is paid off by the massive data parallelism available in the GPU [8]. However, for small tasks, this extra cost might degrade performance.

2.3 Call Re-Vectorization

Introducing an inner dimension of parallelism is desirable to implement irregular algorithms such as graph traversal and recursive sorting. Unfortunately, current abstractions based on warp-synchronous programming or Dynamic Parallelism either compromise efficiency or programmability. To solve this conundrum, we introduce Call Re-Vectorization (CREV), a new programming idiom. Syntactically, CREV is akin to CUDA's dynamic parallelism. Semantically, it avoids the cost of scheduling new kernels.

CREV revisits the concept of **everywhere** (also known as **all** or **unmasked**) blocks to temporarily re-enable inactive threads within divergent regions. Such construction was available in programming languages for SIMD machines, such as C* [26], MPL (*MasPar Programming Language*) [20] or POMPC [15] in the late 1980s and early 1990s, and has made a recent comeback in ISPC [25]. In these languages, an **everywhere** block is executed by every processing element, regardless of its divergent state. At the end of that block, threads are sent back to their original state.

The **everywhere** block is a low-level construct to support the implementation of CREV; however, programmers do not deal with it directly – this is the task of the code generator. Algorithm 4 shows how Algorithm 1 looks like, once implemented using CREV. Programmers use the **crev** keyword at line 6 to re-vectorize functions. CREV maintains a stack of thread states to track execution contexts, thus supporting nested calls of r-functions. In terms of performance, a call to a function using the **crev** directive is equivalent to a regular function call – unlike the implementation of dynamic parallelism in CUDA, for instance. Thus, we favour the use of CREV for fine grain nested parallelism. Section 3 will explain the nitty-gritties behind the CREV directive.

3. Semantics of CREV

This section presents the semantics of Call Re-Vectorization. First, in Section 3.1, we state informally key features of CREV. In Section 3.2, we introduce μ-SIMD, a low-level programming language with a set of primitives that lets us

Algorithm 4: SIMD Book Filter using CREV

```
1 W ← warp size; t_id ← thread index;
2 Function bFilter(mtx b_i, mtx b_o, vec p, int N)
3     for k ← t_id to num_lines(b_i)-1 step W do
4         l ← b_i[k];
5         if memcmp(l, p, N) = 0 then
6             crev memcpy_crev(l, b_o[k], N);
7 Function memcpy_crev(vec s, vec d, int N)
8     for k ← t_id to N − 1 step W do
9         d[k] ← s[k];
```

implement CREV. In Section 3.3, we show how to implement the **crev** high-level construct using the building blocks available in μ-SIMD. Finally, in Section 3.4, we use our semantics to state some properties of CREV. Before we dive in these details, Example 3.1 arms the reader with some intuition on how CREV works.

Example 3.1 *A function is called with the* ***crev*** *prefix to indicate that every thread, whether enabled or disabled, should execute the function. Every thread should execute the r-function multiple times if multiple enabled threads in the warp call it. For instance, if the warp size is 32 and 7 threads are enabled when the program flow hits line 6 in Algorithm 4, all 32 threads execute* `memcmp_crev` *7 times. In each case, the 32 threads temporarily take on the local state of the active thread that they are helping. Once done, these workers all get their local state restored.*

3.1 The Cornerstones of CREV

CREV is defined as follows: for each active thread that reaches a call tagged with **crev**, we execute the target function once, forwarding global parameters (scalars) and extracting private ones per active thread (vectors). This principle of Call Re-Vectorization lays on three pillars: *thread re-activation*, *SIMD function call* and *data distribution*.

Thread Re-Activation. CREV does not lead to the creation of new threads. A function invoked via a CREV call is executed by every thread that is part of a warp, regardless of its threads' state. As mentioned in Section 2, a thread might be inactive due to divergences. However, dormant threads are re-activated to perform work. The former state of the thread is saved into the context stack, used for divergence management. On software context stack implementations such as used on AMD GPUs and Intel AVX-512 platforms, this operation is performed entirely in software. For platforms with hardware context stack implementations, like NVIDIA GPUs, it will require a new machine instruction.

SIMD Function Call. Multi-thread and SIMD languages have different definitions of function calls. In MT, a function call is only performed by active threads. Only register lanes that correspond to active threads are saved. The other threads are guaranteed to stay inactive during execution of the function and need no context save. Although each thread conceptually has its own private call stack, the call stacks of a warp are typically synchronized for performance reasons and to allow the sharing of a single scalar stack pointer for a warp. Implementations of SIMD languages, on the other hand, save whole vector registers on function calls, keeping one stack pointer per warp. Unlike regular MT functions, r-functions follow an SIMD application binary interface. This ensures that all registers in-use are saved before being overwritten inside the function, including lanes of threads that were inactive. Because no threads are created, context switch is similar to the cost of invoking a new function.

branch if zero	`bz` v, l
unconditional branch	`jmp` l
branch if thread previously active	`jmp_mask` t_{id}, l
write to shared memory	$\uparrow v_x = v$
read from shared memory	$v = \downarrow v_x$
binary operations	$v_1 = o_1 \oplus o_2$
copy	$v = o$
shuffle data between lanes	`shfl`(v, v_{lane})
synchronization barrier	`sync`
halt the machine	`stop`
begin `everywhere` *block*	`everywhere`
end `everywhere` *block*	`end_everywhere`

Figure 1. μ-SIMD instruction set. Operands (o) can be either variables or integer constants.

Labels (L)	$::= l \in \mathbb{N}$
Constants (C)	$::= c \in \mathbb{N}$
Variables (V)	$::= T_{id} \cup \{v_1, v_2, \dots\}$
Instructions (I)	::= Figure 1
Active Threads	$\Theta \subset \mathbb{N}$
Local Memory	$\sigma \subset V \mapsto \mathbb{Z}$
Local Memory Bank	$\beta \subset T_{id} \mapsto \sigma$
Shared Memory	$\Sigma \subset \mathbb{N} \mapsto \mathbb{Z}$
Synch Stack	$\Pi \subset (L \times \Theta \times L \times \Theta \times \Pi)$
Context Stack	$\Lambda \subset (\Theta \times \Pi \times \Lambda)$
Program	$P \subset L \mapsto I$
Program Counter	$pc \in \mathbb{N}$

Figure 2. The state of μ-SIMD machine is a septuple $M(\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc)$. Θ is the set of active threads. A thread $t \in \Theta$ has a local memory σ, accessible through a memory bank β. Threads communicate through shared memory Σ. The stack Π tracks control flow divergences. A key component of Call Re-Vectorization is the thread stack Λ. The program counter, *pc*, keeps track of the next instruction $\iota \in P$ to be executed. The program P is a linear sequence of instructions. Although it never changes, we include it as state for convenience.

Data Distribution. Each formerly active warp thread is serialized to have a full warp operate on its data. As in the warp-synchronous `memcpy` example (Algorithm 2), this requires extracting and broadcasting each thread's register lane. Data distribution will be later detailed in Algorithm 5.

3.2 Low-Level Semantics

We formalize the notion of Call Re-Vectorization on top of a core language, μ-SIMD. This language provides the low-level constructs necessary to implement r-functions. Most of the syntax of μ-SIMD comes from Sampaio *et al.* [28], who, in turn, have reused ideas from Bougé *et al.* [2] and Farrell *et al.* [10]. A μ-SIMD program is a sequence of instructions indexed by a *pc*. Figure 1 shows μ-SIMD's syntax.

$$\textbf{split}(\Theta, \beta, v) = (\Theta_0, \Theta_n) \textbf{ where}$$
$$\Theta_0 = \{t \mid t \in \Theta \textbf{ and } \beta[t] = \sigma_t \textbf{ and } \sigma_t[v] = 0\}$$
$$\Theta_n = \{t \mid t \in \Theta \textbf{ and } \beta[t] = \sigma_t \textbf{ and } \sigma_t[v] \neq 0\}$$

$$\textbf{push}([], \Theta_n, pc, l) = [(pc, [], l, \Theta_n)]$$
$$\textbf{push}((pc', [], l', \Theta'_n) : \Pi, \Theta_n, pc, l) = \Pi' \textbf{ if } pc \neq pc'$$
$$\textbf{where } \Pi' = (pc, [], l, \Theta_n) : (pc', [], l', \Theta'_n) : \Pi$$
$$\textbf{push}((pc, [], l, \Theta'_n) : \Pi, \Theta_n, pc, l) = (pc, [], l, \Theta_n \cup \Theta'_n) : \Pi$$

Figure 3. Auxiliar functions used to define μ-SIMD. `split` is a filter, dividing *threads* into two divergent sets (Θ_0 and Θ_n). Auxiliary function `push` updates the synchronization stack Π due to control flow divergences.

Operational Semantics. The state M of a program is a tuple $(\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc)$, as described in Figure 2. Threads are uniquely identified by a natural t_{id}, having a local memory $\beta[t_{id}]$, and sharing a global memory Σ. Memory is vectorized, thus, a local address v denotes a vector of variables $v \in \beta[t_{id}]$; hence, each thread sees its private version of v.

To formalize the semantics of μ-SIMD, we use the auxiliary functions shown in Figure 3. The semantics of μ-SIMD is given by Figures 4 and 5. The former shows the behavior of instructions that change the program's control flow; the latter shows the behavior of logic and arithmetic instructions. The result of executing a control flow instruction is a triple (Θ, β, Σ). The interface between Figure 4 and Figure 5 is performed by Rules IT and TL. The result of executing an arithmetic or logic instruction is a pair (β, Σ), i.e., they only update the program memory.

The semantics of control flow divergences. To simulate the effect of divergences, μ-SIMD has a stack Π. Each element in Π is a tuple $(l_{id}, \Theta_{done}, l_{next}, \Theta_{todo})$, which indicates the point where divergent threads must re-converge. A new tuple is pushed onto Π due to a conditional branch, located at l_{id}, that has caused a divergence, as described by Rules BT, BF and BD, in Figure 4. Θ_{done} is the set of threads that have reached the synchronization point. Θ_{todo} is the set of threads waiting to execute. These threads, once active, will resume execution at label l_{next}. The stack is popped by instructions `sync`, whose behavior is given by Rules SS and SP.

The Thread Stack. To implement CREV, we have added a *thread stack* Λ to μ-SIMD. This stack is fundamental to the implementation of **everywhere** blocks. Λ holds pairs (Θ, Π). Figure 4 shows that instructions `everywhere` (Rule EB) push elements onto Λ, and instructions `end_everywhere` (Rule EE) pop it. The first element in this tuple is the set of threads active immediately before the execution of an **everywhere** block. The second element is the divergence stack, also in the state before the execution of the last **everywhere** block traversed by the program flow. In Rule EB (Fig. 4), Θ_{all} represents all the threads available in a warp.

$$\text{(SP)}\quad \frac{P[pc] = \texttt{stop}}{(\Theta, \beta, \Sigma, \emptyset, \Lambda, P, pc) \to (\Theta, \beta, \Sigma)}$$

$$\text{(JP)}\quad \frac{P[pc] = \texttt{jmp}\ l \qquad (\Theta, \beta, \Sigma, \Pi, \Lambda, P, l) \to (\Theta', \beta', \Sigma')}{(\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc) \to (\Theta', \beta', \Sigma')}$$

$$\text{(BT)}\quad \frac{P[pc] = \texttt{bz}\ v, l \quad \textbf{split}(\Theta, \beta, v) = (\Theta, \emptyset) \quad \textbf{push}(\Pi, \emptyset, pc, l) = \Pi' \quad (\Theta, \beta, \Sigma, \Pi', \Lambda, P, l) \to (\Theta', \beta', \Sigma')}{(\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc) \to (\Theta', \beta', \Sigma')}$$

$$\text{(BF)}\quad \frac{P[pc] = \texttt{bz}\ v, l \quad \textbf{split}(\Theta, \beta, v) = (\emptyset, \Theta) \quad \textbf{push}(\Pi, \emptyset, pc, l) = \Pi' \quad (\Theta, \beta, \Sigma, \Pi', \Lambda, P, pc + 1) \to (\Theta', \beta', \Sigma')}{(\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc) \to (\Theta', \beta', \Sigma')}$$

$$\text{(BD)}\quad \frac{P[pc] = \texttt{bz}\ v, l \quad \textbf{split}(\Theta, \beta, v) = (\Theta_0, \Theta_n) \quad pc' = pc + 1 \quad \textbf{push}(\Pi, \Theta_n, pc, l) = \Pi' \quad (\Theta_0, \beta, \Sigma, \Pi', \Lambda, P, pc') \to (\Theta', \beta', \Sigma')}{(\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc) \to (\Theta', \beta', \Sigma')}$$

$$\text{(BA)}\quad \frac{P[pc] = \texttt{jmp\_mask}\ T_{id}, l \qquad T_{id} \in \Theta' \quad (\Theta, \beta, \Sigma, \Pi, (\Theta', \Pi') : \Lambda, P, l) \to (\Theta'', \beta', \Sigma')}{(\Theta, \beta, \Sigma, \Pi, (\Theta', \Pi') : \Lambda, P, pc) \to (\Theta'', \beta', \Sigma')}$$

$$\text{(BI)}\quad \frac{P[pc] = \texttt{jmp\_mask}\ T_{id}, l \qquad T_{id} \notin \Theta' \quad (\Theta, \beta, \Sigma, \Pi, (\Theta', \Pi') : \Lambda, P, pc + 1) \to (\Theta'', \beta', \Sigma')}{(\Theta, \beta, \Sigma, \Pi, (\Theta', \Pi') : \Lambda, P, pc) \to (\Theta'', \beta', \Sigma')}$$

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

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

$$\text{(EB)}\quad \frac{P[pc] = \texttt{everywhere} \quad (\Theta_{all}, \beta, \Sigma, \emptyset, (\Theta, \Pi) : \Lambda, P, pc + 1) \to (\Theta', \beta', \Sigma')}{(\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc) \to (\Theta', \beta', \Sigma')}$$

$$\text{(EE)}\quad \frac{P[pc] = \texttt{end\_everywhere} \quad (\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc + 1) \to (\Theta', \beta', \Sigma')}{(\_, \beta, \Sigma, \emptyset, (\Theta, \Pi) : \Lambda, P, pc) \to (\Theta', \beta', \Sigma')}$$

$$\text{(IT)}\quad \frac{P[pc] = \iota \qquad \iota \neq \text{Control Flow Instruction} \quad (\Theta, \beta, \Sigma, \Theta_{mask}, \iota) \to (\beta', \Sigma') \qquad pc' = pc + 1 \quad (\Theta, \beta', \Sigma', \Pi, (\Theta_{mask}, \Pi') : \Lambda, pc') \to (\Theta', \beta'', \Sigma'')}{(\Theta, \beta, \Sigma, \Pi, (\Theta_{mask}, \Pi') : \Lambda, P, pc) \to (\Theta', \beta'', \Sigma'')}$$

Figure 4. Semantics of μ-SIMD's control flow instructions.

The thread stack lets us represent an unbounded number of different thread contexts; hence, programs might contain an

$$\text{(MM)}\quad \frac{\Sigma(v) = c}{\Sigma \vdash v = c} \qquad \text{(TL)}\quad \frac{(t, \beta, \Sigma, \Theta_{mask}, \iota) \to (\sigma_t, \Sigma') \qquad (\Theta, \beta \setminus [\beta[t] \mapsto \sigma_t], \Sigma', \Theta_{mask}, \iota) \to (\beta", \Sigma")}{(\{t\} \cup \Theta, \beta, \Sigma, \Theta_{mask}, \iota) \to (\beta", \Sigma")}$$

$$\text{(MT)}\quad t, \beta \vdash t_{id} = t \qquad \text{(BP)}\quad \frac{t, \beta \vdash v_2 = c_2 \qquad t, \beta \vdash v_3 = c_3 \qquad \beta[t] = \sigma_t \qquad c_1 = c_2 \oplus c_3}{(t, \beta, \Sigma, \_, v_1 = v_2 \oplus v_3) \to (\sigma_t \setminus [v_1 \mapsto c_1], \Sigma)}$$

$$\text{(MV)}\quad \frac{\beta[t] = \sigma_t \qquad \sigma_t(v) = c}{t, \beta \vdash v = c} \qquad \text{(SI)}\quad \frac{t, \beta \vdash v_1 = c_1 \qquad t, \beta \vdash v_{lane} = c_{lane} \qquad \beta[t] = \sigma_t \qquad c_{lane} \notin \Theta_{mask}}{(t, \beta, \Sigma, \Theta_{mask}, \texttt{shfl}(v_1, v_{lane})) \to (\sigma_t \setminus [v_1 \mapsto \_], \Sigma)}$$

$$\text{(SV)}\quad \frac{t, \beta \vdash v_1 = c_1 \qquad t, \beta \vdash v_{lane} = c_{lane} \qquad \beta[t] = \sigma_t \qquad c_{lane} \in \Theta_{mask} \qquad \beta[c_{lane}] = \sigma_{lane} \qquad \sigma_{lane}(v_1) = c_2}{(t, \beta, \Sigma, \Theta_{mask}, \texttt{shfl}(v_1, v_{lane})) \to (\sigma_t \setminus [v_1 \mapsto c_2], \Sigma)}$$

$$\text{(CT)}\quad \frac{\beta[t] = \sigma_t}{(t, \beta, \Sigma, \_, v = c) \to (\sigma_t \setminus [v \mapsto c], \Sigma)} \qquad \text{(AS)}\quad \frac{t, \beta \vdash v' = c \qquad \beta[t] = \sigma_t}{(t, \beta, \Sigma, \_, v = v') \to (\sigma_t \setminus [v \mapsto c], \Sigma)}$$

$$\text{(LD)}\quad \frac{t, \beta \vdash v_x = c_x \qquad \beta[t] = \sigma_t \qquad \Sigma \vdash c_x = c}{(t, \beta, \Sigma, \_, v =\downarrow v_x) \to (\sigma_t \setminus [v \mapsto c], \Sigma)} \qquad \text{(ST)}\quad \frac{t, \beta \vdash v_x = c_x \qquad t, \beta \vdash v = c \beta[t] = \sigma_t}{(t, \beta, \Sigma, \_, \uparrow v_x = v) \to (\sigma_t, \Sigma \setminus [c_x \mapsto c])}$$

Figure 5. Semantics of arithmetic, logic and data-related instructions. Rule TL loops over every thread $t \in \Theta$, and for each one of them, executes instruction ι. No assumption can be made on the order in which instructions run.

	Instructions
	v0 = ↓tid
	v1 = (v0 == 0)
	bz v1, Done
	v2 = 4 * (tid + 1)
	everywhere
	v8 = 0
Loop	jmp_mask v8, Call
	jmp Next
Call	v3 = shfl(v2, v8)
	v4 = v3 + tid
	v5 = ↓v4
	v6 = v5 + 1
	↑v4 = v6
Next	v8 = v8 + 1
	v7 = (v8 == 4)
	bz v7, Loop
	end_everywhere
	↑tid = 1
Done	sync

Shared Memory

Address	0	1	2	3
Contents	0	1	1	0
Address	4	5	6	7
Contents	2	1	3	4
Address	8	9	10	11
Contents	1	5	6	1
Address	12	13	14	15
Contents	2	3	1	7
Address	16	17	18	19
Contents	1	3	4	0

Private Memory

Address	v0	v1	v2	v3
Contents	*	*	*	*
Address	v4	v5	v6	v7
Contents	*	*	*	*
Address	v8			
Contents	*			

Figure 6. Program written in μ-SIMD, plus its initial state.

arbitrary number of nested **everywhere** blocks. After executing the instruction `end_everywhere`, threads previously inactive will go back into sleeping mode. In other words, after `end_everywhere`, the pair (Θ, Π) at the top of Λ is popped, and the diverging configuration Π becomes part of the current state of threads. If necessary to check if a thread is active due to an **everywhere** block, then μ-SIMD provides a conditinal `jmp_mask`. The statement `jmp_mask`(t_{id}, l) will divert execution to l if t_{id} is active in the mask at Λ's top. Example 3.2 illustrates the behavior of these instructions.

Example 3.2 *Figure 6 shows a program written in μ-SIMD. We assume $|\Theta_{all}| = 4$. This program increments a 4×4 matrix; however, line i is incremented only if $\Sigma[t_{id}] = 0$. The figure shows the initial state of the shared (Σ) and local memory of each thread (σ). The initial state of the variables in the local memory is immaterial for this example. Figure 7 shows a trace of the execution of the program, given its initial state. Only threads $t_{id} = 0$ and $t_{id} = 3$ will enter the **everywhere** section, because $\Sigma[0] = \Sigma[3] = 0$. Nevertheless, all the four threads will execute the commands within that block. Instruction v3 = shfl(v2, v8) lets each thread read into v3 the value of v2 seen by thread v8.*

3.3 High-Level Semantics

The μ-SIMD assembly gives us the primitive building blocks to implement CREV in higher-level languages. As a proof of concept, we have implemented CREV onto ISPC, using instructions of ISPC that are equivalent to those seen in μ-SIMD. By focusing on an abstract notation, μ-SIMD, instead of on a concrete language, such as ISPC, we claim generality: CREV can be implemented in any environment that supports our notions of **everywhere** and shuffle. In this section we show how to implement the **crev** modifier, which marks a function call as an r-function. For simplicity, our high-level language provides only syntax to declare and invoke functions. A function declaration consists of a *name* f, plus a list of *formal parameters*, e.g.: $f(T\ p_1, \ldots, T\ p_n)$. We let T denote a *type modifier*, which can be either *uniform* or *varying*. We have borrowed this notation from ISPC. Other programming languages have different ways to express these modifiers. For instance, in CUDA we have *shared* and *global* allocation filling the role of ISPC's uniform variables.

Figure 8 shows the code that we produce for a r-function call $f(a_1, \ldots, a_n)$, where each $a_i, 1 \leq i \leq n$ is an actual argument of f. Such an r-function call will trigger up to $|\Theta|$ executions of f, one for each active thread $t \in \Theta$. The test in lines 3 or 8 in Figure 8 are used to single out the function

	Instructions	Var	Tid 0	1	2	3	Tid 0	1	2	3
	v0 = ↓tid	v0	0	1	1	0	✓	✓	✓	✓
	v1 = (v0 == 0)	v1	1	0	0	1	✓	✓	✓	✓
	bz v1, Done		F	T	T	F	✓	✓	✓	✓
	v2 = 4 * (tid + 1)	v2	4	*	*	16	✓	•	•	✓
	everywhere						✓	•	•	✓
	v8 = 0	v8	0	0	0	0	✓	✓	✓	✓
Loop	jmp_mask v8, Call		T	T	T	T	✓	✓	✓	✓
	jmp Next		F	F	F	F	✓	✓	✓	✓
Call	v3 = shfl(v2, v8)	v3	4	4	4	4	✓	✓	✓	✓
	v4 = v3 + tid	v4	4	5	6	7	✓	✓	✓	✓
	v5 = ↓v4	v5	2	1	3	4	✓	✓	✓	✓
	v6 = v5 + 1	v6	3	2	4	5	✓	✓	✓	✓
	↑v4 = v6						✓	✓	✓	✓
Next	v8 = v8 + 1	v8	1	1	1	1	✓	✓	✓	✓
	v7 = (v8 == 4)	v7	0	0	0	0	✓	✓	✓	✓
	bz v7, Loop		T	T	T	T	✓	✓	✓	✓
Loop	jmp_mask v8, Call		F	F	F	F	✓	✓	✓	✓
	jmp Next		T	T	T	T	✓	✓	✓	✓
Next	v8 = v8 + 1	v8	2	2	2	2	✓	✓	✓	✓
	v7 = (v8 == 4)	v7	0	0	0	0	✓	✓	✓	✓
	bz v7, Loop		T	T	T	T	✓	✓	✓	✓
Loop	jmp_mask v8, Call		F	F	F	F	✓	✓	✓	✓
	jmp Next		T	T	T	T	✓	✓	✓	✓
Next	v8 = v8 + 1	v8	3	3	3	3	✓	✓	✓	✓
	v7 = (v8 == 4)	v7	0	0	0	0	✓	✓	✓	✓
	bz v7, Loop		T	T	T	T	✓	✓	✓	✓
Loop	jmp_mask v8, Call		T	T	T	T	✓	✓	✓	✓
	jmp Next		F	F	F	F	✓	✓	✓	✓
Call	v3 = shfl(v2, v8)	v3	16	16	16	16	✓	✓	✓	✓
	v4 = v3 + tid	v4	16	17	18	19	✓	✓	✓	✓
	v5 = ↓v4	v5	1	3	4	0	✓	✓	✓	✓
	v6 = v5 + 1	v6	2	4	5	1	✓	✓	✓	✓
	↑v4 = v6						✓	✓	✓	✓
Next	v8 = v8 + 1	v8	4	4	4	4	✓	✓	✓	✓
	v7 = (v8 == 4)	v7	1	1	1	1	✓	✓	✓	✓
	bz v7, Loop		F	F	F	F	✓	✓	✓	✓
	end_everywhere						✓	•	•	✓
	↑tid = 1						✓	•	•	✓
Done	sync						✓	✓	✓	✓

Figure 7. Execution trace of the program in Figure 6. Column **Var** shows contents of last variable assigned. T indicates branch taken; F indicates otherwise. The symbol • marks inactive threads. For the syntax of instructions, we refer the reader to Fig. 1; for their semantics, Figs. 4 and 5.

```
1          everywhere                    ;; begin CREV
2          i = 0                         ;; Loop counter
3   loop:  jmp_mask i, call
4          jmp next                      ;; Skip idle threads
5   call:  extract(t^n, p^n, a^n, i)     ;; Algorithm 5
6          "call" f                      ;; function call
7   next:  i = i + 1
8          bnz(i ≠ W) loop
9          end_everywhere                ;; end CREV
```

Figure 8. Low-level code produced to call r-function f.

invocation performed by each thread. A different call will happen due to each $handle_t$ label. In another dimension of parallelism, each function call will be executed by Θ_{all} threads, due to the **everywhere** block at lines 10 and 13. Thus, we might have up to Θ^2_{all} computations.

Algorithm 5: Data distribution

```
1 Function declaration: f(p_1, ..., p_n);
2 Function call: f(t_1 a_1, ..., t_n a_n);
3 Function extract(t^n, p^n, a^n, i)
4     for k ∈ 1...n do
5         if t_k == uniform then
6             p_k = a_k;
7         if t_k == varying then
8             shfl(a_k, i);
```

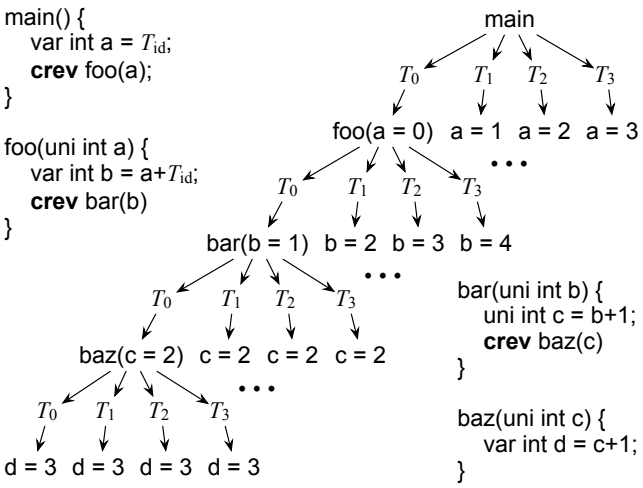

Figure 9. A program written in ISPC, and the tree showing function calls for T_0.

Algorithm 5 generates code that implements data distribution. Data distribution determines how actual parameters are bound to formal parameters, given that actual parameters can have one of two types: *uniform* or *varying*. By construction, r-functions have only uniform parameters. The loop in line 4 will go over all the function arguments, comparing formal (p) and actual (a) parameters. We let the type of a_i be t_i. If an actual argument is uniform, then parameter passing is trivially implemented as a copy between variables. Line 5 of Algorithm 5 generates code under such circumstance. If an actual parameter has type varying, then we generate code to perform a broadcast, as seen in line 7 of Algorithm 5.

Example 3.3 *The program in Figure 9 shows three functions called via* ***crev****. We are assuming an architecture with four SIMD lanes, i.e.,* $\Theta_{all} = \{T_0, T_1, T_2, T_3\}$*. When foo is invoked, the value of a, main's local variable, is broadcasted to foo's formal parameter. Thus,* T_0 *sees foo(0),* T_1 *sees foo(1), etc. When* T_0 *calls bar from foo, the same behavior is observed. However, when* T_0 *calls baz from bar, all the four threads activated into this context see baz(2), because baz receives a uniform argument. The fact that baz's*

local variable d *is marked as varying is immaterial in this example, as this variable is initialized with uniform values.*

3.4 Properties of CREV

The semantics of CREV, given by μ-SIMD's primitive building blocks, and the translator seen in Section 3.3, lets us establish a few properties that are true about this programming abstraction. In this section we go over a few of these properties. They are valid under the assumption that programs are *well-formed*. We define well-formed programs below:

Definition 3.4 (Well-Formed Program) *A μ-SIMD program is well-formed if any occurrence of an* `everywhere` *instruction at label l_1 is matched by an occurrence of an* `end_everywhere` *instruction at label l_2, and these two labels are* control equivalent.

Definition 3.4 borrows the concept of control equivalence from Ferrante *et al.* [11]. Two points, l_1 and l_2, in a program's control flow graph are said to be control equivalent if l_1 dominates l_2, and l_2 post-dominates l_1. We say that l_1 dominates l_2 if, and only if, any path from the root of the CFG to l_2 must cross l_1. Dually, l_2 post-dominates l_1 if, and only if, any path from l_1 to the end of the CFG must cross l_2. Our translator produces well-formed programs, as long as the program flow cannot leave a function through points other than its return address.

Theorem 3.5 (Well-Formed Translation) *The translator of Figure 8 produces well-formed programs.*

> **Proof:** This result follows trivially from the fact that an **everywhere** block surrounds only Algorithm 5 and the r-function. Well-formedness holds as long as none of these routines let the program flow escape the enclosing `end_everywhere` instruction. This implies that the r-function cannot throw exceptions, for instance. □

Composability. CREV allows the nesting of **everywhere** blocks. Composition happens due to nested function calls. The thread stack Λ ensures that the last invoked r-function will be the first to remove pending computation. In what follows, we visit three consequences of this property.

Composition is multiplicative. An **crev** call will put all the warp threads in active mode. By coupling this observation with *composibility*, we have that, in the absence of divergences, a sequence of n nested **crev** calls will create $|\Theta_{all}|^N$ tasks. Notice that CREV produces new tasks, but not new threads: we still have only $|\Theta_{all}|$ threads to solve these tasks.

Commutativity. The translator of Figure 8 calls an r-function in a lexicographic order defined by thread identifiers. However, μ-SIMD's primitives do not impose any order on the threads pushed onto Λ. Therefore, the multiple SIMD calls of an r-function can be handled in any order.

Synchronization parity. There is no distinction between the top level of parallelism and the inner level of parallelism with regards to the synchronization primitive. In other words, divergences are handled transparently by the synchronization stack Π, and, from a synchronization standpoint, it is not possible to tell if execution exists within the context of an r-function or not. To ensure this property, μ-SIMD's `everywhere` instruction pushes onto Λ, together with the set of active threads, the divergent state Π.

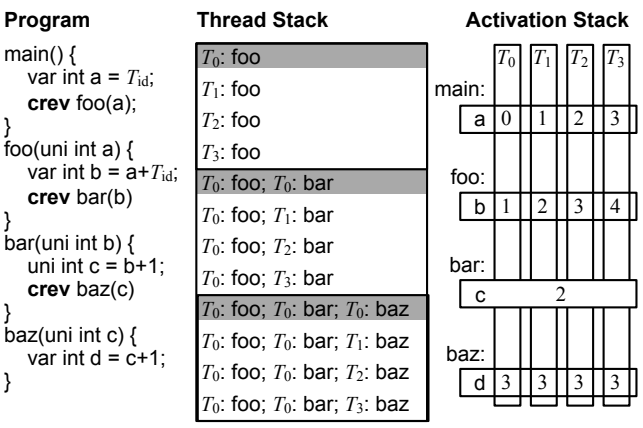

Figure 10. Example of three nested calls to r-functions. Calls currently in the activation stack are highlighted.

The interplay between CREV and nested function calls. The implementation of CREV does not interfere with the implementation of function calls. Programming languages that support recursion use a structure known as *activation stack* to manage function calls. Entries in the activation stack are called *activation records*, and they store functions' local variables, return address, arguments, etc. Upon invocation, the activation record of a function is pushed onto the activation stack. For each thread pushed onto the thread stack there will exist one activation record on the activation stack. The multiplicative nature of CREV also implies on a multiplication of activation records. Therefore, n nested r-calls will generate $|\Theta_{all}|^n$ activation records; however, the maximum depth of the activation stack is still $n+1$: activation records owned by different threads will not exist simultaneously.

Example 3.6 *Figure 10 reuse the program from Example 3.3 to illustrate these points. Again, we assume $|\Theta_{all}| = 4$. Thus, three non-divergent nested r-calls will create $4\times4\times4\times4 = 256$ tasks. At any time, the thread stack will contain at most $4+4+4+4 = 16$ tasks waiting for execution. The activation stack will contain, at any given point, at most 4 activation records, corresponding to the activation of functions* main, foo, bar *and* baz.

4. Evaluation

To evaluate the ideas presented in this paper, we have implemented CREV on ISPC. We use this implementation to demonstrate that CREV allies the efficiency of warp synchronous programming with the clarity and elegance of dy-

namic parallelism; hence, avoiding the complexity of the former, and the heavy scheduling cost of the latter.

Experimental Setup. Because CREV is a novel concept within ISPC, this compilation framework does not provide benchmarks that use it. Thus, we have re-implemented seven classic algorithms using the new keyword **crev**. Our seven benchmarks are: (bk) Book Filter (Algorithm 1); (sm) String Matching; (bf) Bellman-Ford [1]; (df) Depth-First Search; (le) Leader Election; (qs) quick-sort; and (ms) merge-sort.

Runtime Environment. We have implemented CREV onto ISPC v 1.9.1, and have used it to target a 6-core 2.00 GHz Intel Xeon E5-2620 CPU with 8-wide AVX vector units, running Linux Ubuntu 12.04 3.2.0. This running environment gives us warps with eight threads, e.g., $|\Theta_{all}| = 8$.

The Competing Approaches. We compare four different ways to implement our benchmarks.

- Seq: serial implementation of each algorithm, as defined in Cormen's book [5]. String-matching was implemented after the Knuth–Morris–Pratt (KMP) [19] algorithm.
- Par: warp synchronous implementation using constructs available in the ISPC language, but without CREV.
- Launch: dynamic parallelism, implemented via the **launch** keyword, which starts a new PThread per function call.
- CREV: the implementation of the algorithms using the ideas introduced in this paper.

The Seq version of each benchmark is implemented in C++, and is compiled with clang version 3.7.1, with the optimization flags -O2/-std=c++11. Benchmarks in the other three categories (Par, Launch and CREV) are compiled with ISPC. The Launch and CREV implementations look the same, except for the keyword that precedes function calls: **launch** in the first case, and **crev** in the second. The implementations in the Par group are different: they do not contain function calls within divergent regions, to ensure correctness.

Implementations are not available for all the programs. Even though it is trivial to implement quick-sort or merge-sort with dynamic parallelism or with CREV, the craft of a warp synchronous implementation of them is not obvious; hence, we omit them. Additionally, we omit a sequential implementation of book filter, because this problem does not have a canonical, textbook-like, solution. Notice, however, that this algorithm reuses two r-functions: string match (sm), and `memcpy` (Algorithm 2). Thus, we show results for sequential string matching, but not for sequential book filter.

How to read our results. Results are measured in millions of execution cycles, as reported by ISPC's testing environment. Numbers are the average of five, out of six samples. We have removed the first, to avoid cold-start discrepancies. The reader must bear four observations in mind, when analyzing our results: (i) speedups of CREV over pure ISPC (Par) are due to the better load distribution that CREV accomplishes by transporting work to inactive threads; (ii) the large slowdowns observed with Launch are due to the heavy cost of scheduling millions of new Posix threads to perform small chunks of work. (iii) CREV's slowdowns are due to the boilerplate code necessary to serialize threads, before invoking r-functions; (iv) we are comparing against an industrial-strength compiler; hence, speedups tend to be modest.

Pg	Seq	Par	Launch	CREV
bk	×	8,530.99	7,857.98	7,405.17
sm	6,649.28	3,576.14	393,166.27	2,737.94
bf	141,088.73	493,619.69	•	529,856.06
df	3,754.10	3,786.26	•	3,790.44
le	4,054.66	3,983.09	5,272.92	3,984.79
qs	2.87	×	204.28	2.87
ms	7.30	×	104.98	4.11

Figure 11. Execution time, in millions of cycles. We use × to indicate that the implementation of a benchmark is not available, and • to indicate that the implementation of the benchmark does not run successfully until completion.

4.1 Results and Discussion

Figure 11 summarizes our results. The table shows runtime, measured as number of execution cycles (in millions), for all the implementations that we have of the seven different benchmarks. These numbers are produced with the largest inputs that we have for each benchmark:

- bk: 10K strings of 0s and 1s, each with 20K bits, and random target pattern with 16 bits.
- sm: 256MB taken from books available in the Project Gutenberg, and a target pattern with 16 characters.
- bf: random Erdös-Rényi [9] graph with 2,048 nodes and 80% edge probability.
- df,le: 8-ary complete tree, depth 5 (root + five full levels).
- qs,ms: 16K random integers in the range $[0, 100000)$.

Figure 11 shows that ISPC, with or without CREV, is competitive against mainstream compilers: Seq is the best C implementation that we could produce of each algorithm; furthermore, these programs are compiled with clang -O2 (which was faster than clang -O3 in this experiment). Nevertheless, CREV and Par outperform these implementations in several cases, although they are not embarrassingly parallel. The table also reveals that Launch is sometimes orders of magnitude slower than the other approaches, being the fastest in only one case: bk. In bf and df, the excessive number of threads created by Launch forced earlier termination.

CREV usually outperforms Par by a small margin. However, the main benefit of the former over the latter is not performance. Rather, it is readability. The Launch and the CREV versions of each benchmark are exactly the same, except that whereas in one case we prepend the call of functions with the launch keywork, in the other we use **crev**. On the other hand, the Par implementations are very different, because they cannot call functions within potentially divergent regions. To circumvent this restriction, the five Par benchmarks forgo recursion. As an example, the ISPC ver-

sion of depth-first search uses a stack of tasks, instead of performing recursive functions to traverse the graph.

Algorithm 6: Pattern matching: CREV vs. Naïve

```
1  P ← pattern; T ← target text;
2  W ← warp size; t_id ← thread index;
3  Function memcmp(Offset k)
4      m ← True;
5      for i ← t_id to |P| do
6          if P[i] ≠ T[i + k] then m ← False ;
7      if all (m = True) then Found(k) ;
8  Function StringMatch
9      for i ← t_id to (|T| − |P|) step W do
10         if P[0] = T[i] then crev memcmp(i) ;
11 Function ParStringMatch
12     for i ← t_id to (|T| − |P|) step W do
13         j ← 0; k ← i;
14         while j < |P| and P[j] = T[k] do
15             j ← j + 1; k ← k + 1;
16         if j = |P| then Found(k) ;
```

As another example, Algorithm 6 shows the CREV-based implementation of string matching. This is a warp-synchronous implementation of parallel matching: each thread t_{id} tries to match P at positions $T[t_{id} + n \times W]$, where $n \leq |T|$, and W is the warp size. Thus, in the best scenario, runtime is divided by W. This implementation is irregular: divergences might happen at lines 6 and 10. Each call to `memcmp`[4] will commence a CREV sequence of computations. Figure 12 compares our implementation, seen in Algorithm 6 (`StringMatch`) against the equivalent Par version, which does not perform any function call.

The Impact of Data on Runtime. The numbers seen in Figure 11 are input dependent. However, the overall conclusions remain the same, once we feed our benchmarks with inputs of different sizes. Figure 12 compares the runtime different implementations of string-matching and leader election. In string-matching, we omitted the Launch-based version, as it was too slow. Notice that KMP has lower asymptotic complexity than Algorithm 6. It runs in $O(|T| + |P|)$, whereas Algorithm 6 runs in $O(|T| \times |P|/|W|)$. For this experiment, we searched for prefixes of the pattern "*She had been watching him the la*", of sizes $4, 8, \ldots, 28, 32$ in Jane Austen's book *Pride and Prejudice*, taken from Project Gutenberg[5]. CREV is always faster than `ParStringMatch`, and runs faster than KMP in more than half the cases. CREV beats plain parallelism because it distributes function `memcmp` among the eight available vector lanes. On the other hand, `ParStringMatch` has a potentially long divergent block in line 14. In our best result, observed for patterns of size eight, CREV runs in 44% of the time taken by `ParStringMatch`, and in 40% of the time taken by KMP.

[4] Function `memcmp` is also used at line 5 of Algorithm 1

[5] https://www.gutenberg.org/ebooks/42671

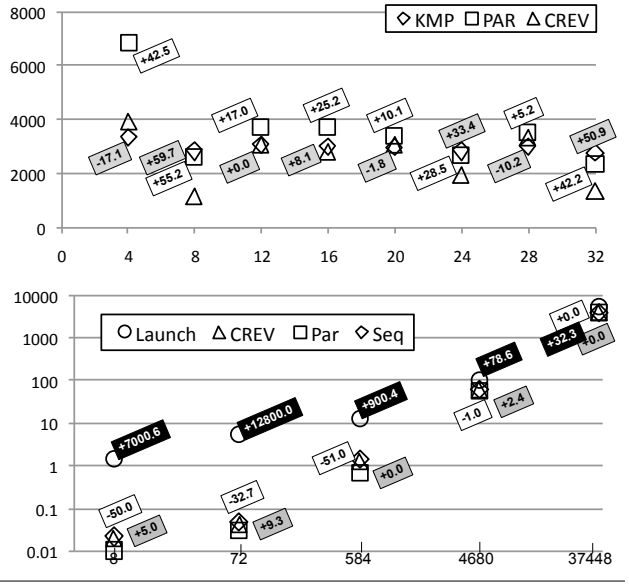

Figure 12. (**Top**) String-Matching. X-axis shows pattern sizes, in number of characters. Target text contains 256MB divided among 5,058,121 lines. (**Bottom**) Leader-Election. X-axis shows graph size, in number of edges. In both charts, Y-axis shows runtime, in millions of cycles. Boxes show percentage of speedup: white is CREV over Par; grey is CREV over Seq, and black is CREV over Launch.

Figure 12 (Bottom) shows a comparison between the four implementations of leader-election. Dynamic parallelism is still the slowest, overall; however, we notice that it tends to catch up with the other approaches for very large inputs, in this benchmark. This behavior is expected: if the amount of parallel work is large, then the greater flexibility and independence of pthreads start paying for the cost of creating them. Nevertheless, depending on how the algorithm is implemented, dynamic parallelism might lead to the creation of a very large number of threads, as the size of the input grows. Due to this observation, we have not been able to run the Launch versions of df and bf for the largest available inputs. On the other hand, in the case of leader-election (le), parallelism is coarser: each thread receives the task of finding strong components in a graph. Thus, even though the number of threads grows as the size of the input increases, this growth is less accentuated than in df and bf.

5. Related Work

GPUs' increasing programmability and decreasing costs have made them very popular for the development of general purpose high performance applications [23]. This popularity has attracted the interest of programming language researchers. Control flow divergences have been a particularly important source of attention. Therefore, the compiler-

related literature contains a vast body of work describing analyses [6, 27, 28, 30] and optimizations [6, 7, 35, 37] that reduce the effects of divergences in GPGPU code. CREV is not a competitor of these analyses and optimizations. On the contrary, Call Re-Vectorization complements such techniques, giving programmers a tool that lets them deal with divergences at the software level. In the rest of this section we touch work that is more closely related to ours.

Everywhere Blocks. The problem of expressing nested SIMD loops in multi-thread style is not new. Some data-parallel programming languages for SIMD computers in the 80's and 90's allow to re-enable temporarily dormant threads. The C* language [26], the Maspar Programming Language [20] and the POMPC language [15] incorporate a control flow construct named either `everywhere` or `all` to this end. We have re-used these instructions to implement CREV. However, these are low-level primitives: they are not programmer-friendly, nor have any interface with function calls. Using **everywhere** directly is difficult, as this abstraction has no knowledge nor control over the state of dormant threads. CREV, on the contrary, is as easy to use as dynamic parallelism. It manages register saves and restores automatically, relieving the programmer from this task.

Warp-level convergence guarantees. Previous work enforce guarantees on where threads converge after control divergences to make warp-synchronous programming safer. For instance, Pharr *et al.* have proposed the *maximal convergence* guarantee [25], and Gaster has proposed a divergence-aware execution model for OpenCL [13]. CREV goes further by actually *enforcing* convergence at arbitrary program points, allowing warp-synchronous functions to be called from divergent sections. To the best of our knowledge, this is the first attempt to provide developers with such possibility.

Grid-level Dynamic Parallelism. Much effort has been spent to reduce the overhead of dynamic parallelism. Alternatives to CUDA Dynamic Parallelism such as DTBL [33], Free Launch [4] and LaPerm [34] reduce sub-kernel launch overhead or improve cache locality. By relying on global schedulers, they allow load-balancing between GPU stream multiprocessors. We are not competing with these efforts, because CREV is not an alternative to dynamic parallelism. CREV is a static code transformation with no dynamic scheduling; hence, it does not create extra parallelism. In other words, we move work to threads that are already in flight, instead of spawning new threads. The main benefit of CREV, when compared to these previous work comes in terms of programmability and efficiency: by supporting composability of multi-thread and SIMD code, we give developers the chance to benefit from efficient warp-synchronous idioms without neither having to deal with primitives like shuffle, vote and population count, nor having to worry about saving the context of threads.

Thread-level divergence aware optimizations. Compilers may reorder computations across loop iterations within each thread to mitigate branch divergence [6, 14, 17, 24]. However, each thread performs the same set of tasks as in the original version, so divergences induced by load unbalance between threads of a warp remains an issue. CREV is a way to deal with irregular programs whose performance divergences hurt. However, CREV is not an optimization implemented by the compiler: programmers must adapt algorithms to use this construct. CREV deals well with divergences because it lets developers balance workload between threads in flight. In other words, it changes the loop structure by distributing iterations across different threads.

6. Final Thoughts

Primitives such as warp vote and shuffle have given experts the possibility of writing efficient SIMD code, by programming from the point of view of one warp. This coding style has been used in CUB and many other CUDA libraries[6]. However, warp-synchronous code does not play well with branch divergence. Most warp-synchronous algorithms require all threads in a warp to be active. This is a problem for the common usage scenario of a simple MT-style CUDA kernel that calls warp-synchronous library functions. It is our vision that the application developer writing the kernel should not be concerned with the internal implementation of library functions, and should be able to call any function inside divergent program regions. To meet the demands of this vision, this paper has introduced the notion of Call Re-Vectorization(CREV). We have described the building blocks necessary to implement CREV. Looking towards compatibility with future hardware, we have proposed low-level primitives with well-defined semantics and a high-level interface, the **crev** idiom, that makes programmer intent explicit. Thus, our notion of CREV does not rely implicitly on current hardware behavior, which might eventually change. We have implemented CREV into ISPC, using Intel instructions, and have shown how to code irregular algorithms in this environment. Our implementations are not only clearer than non-CREV based approaches, but also more efficient, as they balance work among inactive warp threads.

Acknowledgement

This project is supported by the Inria-Fapemig Associate Team *Prospiel* and by CNPq. We thank the members of the *Dragon Nest* group and the PPoPP referees for their insightful comments and suggestions. Example 3.1 has been suggested by one of the referees. Finally, we thank Albert Cohen for shepherding this paper. Our implementation is publicly available at `http://cuda.dcc.ufmg.br/swan/`.

References

[1] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[6] See `https://nvlabs.github.io/cub/`

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

[3] J. Brodman, D. Babokin, I. Filippov, and P. Tu. Writing scalable SIMD programs with ISPC. In *WPMVP*, pages 25–32. ACM, 2014.

[4] G. Chen and X. Shen. Free launch: optimizing GPU dynamic kernel launches through thread reuse. In *Micro*, pages 407–419. ACM, 2015.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill, 2nd edition, 2001.

[6] B. Coutinho, D. Sampaio, F. M. Q. Pereira, and W. M. Jr. Divergence analysis and optimizations. In *PACT*, pages 320–329. IEEE, 2011.

[7] B. Coutinho, D. Sampaio, F. M. Q. Pereira, and W. M. Jr. Profiling divergences in GPU applications. *Concurrency and Computation: Practice and Experience*, 1(10.1002/cpe.285-15):1–15, 2012.

[8] J. DiMarco and M. Taufer. Performance impact of dynamic parallelism on different clustering algorithms. *Modeling and Simulation for Defense Systems and Applications*, 8752(VIII): 87520E–87520E:8, 2013.

[9] P. Erdos and A. Renyi. On random graphs. I. *Publicationes Mathematicae*, 6(1):290–297, 1959.

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

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

[12] M. Garland and D. B. Kirk. Understanding throughput-oriented architectures. *Commun. ACM*, 53:58–66, 2010.

[13] B. Gaster. An execution model for OpenCL 2.0. Technical Report 2014-02, Computer Sciences, 2014.

[14] T. D. Han and T. S. Abdelrahman. Reducing divergence in GPGPU programs with loop merging. In *GPGPU*, pages 12–23. ACM, 2013.

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

[16] S. Jones. Introduction to dynamic parallelism – Invited Talk. In *GPU Technology Conference*, pages 1–33. NVIDIA, 2014.

[17] F. Khorasani, R. Gupta, and L. N. Bhuyan. Efficient warp execution in presence of divergence with collaborative context collection. In *Micro*, pages 204–215. ACM, 2015.

[18] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. Pycuda and pyopencl: A scripting-based approach to gpu run-time code generation. *Parallel Comput.*, 38(3): 157–174, Mar. 2012.

[19] D. E. Knuth, J. H. M. Jr., and V. R. Pratt. Fast pattern matching in strings. *Journal of Computing*, 6(2):323–350, 1977.

[20] MasPar. *MasPar Programming Language (ANSI C compatible MPL) Reference Manual*, 1992.

[21] D. Merrill and A. Grimshaw. High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing. *Parallel Proceessing Letters*, 21 (2):245–272, 2011.

[22] A. Munshi, B. Gaster, T. G. Mattson, J. Fung, and D. Ginsburg. *OpenCL Programming Guide*. Addison-Wesley, 1st edition, 2011.

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

[24] R. Novak. Loop optimization for divergence reduction on GPUs with SIMT architecture. *IEEE Transactions on Parallel and Distributed Systems*, 26(6):1633–1642, 2015.

[25] M. Pharr and W. R. Mark. ISPC: A SPMD compiler for high-performance CPU programming. In *InPar*, pages 1–13. IEEE, 2012.

[26] J. Rose and G. Steele. C*: An extended C language for data parallel programming. In *ICS*, 1987.

[27] D. Sampaio, R. Martins, S. Collange, and F. M. Q. Pereira. Divergence analysis with affine constraints. In *SBAC-PAD*, pages 67–74. IEEE, 2012.

[28] D. Sampaio, R. M. de Souza, S. Collange, and F. M. Q. Pereira. Divergence analysis. *ACM Trans. Program. Lang. Syst.*, 35(4):13, 2013.

[29] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley, 1st edition, 2010.

[30] T. Schaub, S. Moll, R. Karrenberg, and S. Hack. The impact of the simd width on control-flow and memory divergence. *TACO*, 11(4):54:1–54:25, 2015.

[31] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu. Knights Landing: Second-generation Intel Xeon Phi product. *IEEE Micro*, 36(2):34–46, 2016.

[32] J. Wang and S. Yalamanchili. Characterization and analysis of dynamic parallelism in unstructured GPU applications. In *IISWC*, pages 51–60. IEEE, 2014.

[33] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili. Dynamic thread block launch: A lightweight execution mechanism to support irregular applications on gpus. In *ISCA*, pages 528–540. ACM, 2015.

[34] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili. LaPerm: Locality aware scheduler for dynamic parallelism on GPUs. In *ISCA*, 2016.

[35] J. Wu, A. Belevich, E. Bendersky, M. Heffernan, C. Leary, J. Pienaar, B. Roune, R. Springer, X. Weng, and R. Hundt. Gpucc: An open-source gpgpu compiler. In *CGO*, pages 105–116. ACM, 2016.

[36] Y. Yang and H. Zhou. CUDA-NP: realizing nested thread-level parallelism in GPGPU applications. In *PPoPP*, pages 93–106. ACM, 2014.

[37] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly elimination of dynamic irregularities for GPU computing. In *ASPLOS*, pages 369–380. ACM, 2011.