Function Call Re-Vectorization

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

Sylvain Collange  
INRIA – France  
sylvain.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 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.

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 __syncthreads 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 re-

---

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

2 Ex.: see Trove at https://github.com/bryancatanzaro/trove.
activating 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 introduce 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]. 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 µSIMD, 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.

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 \( \text{memcmp} \), and memory copying is done by \( \text{memcpy} \). 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

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( W \leftarrow \text{warp size}; t_{id} \leftarrow \text{thread index}; )</td>
</tr>
<tr>
<td>2</td>
<td>Function ( \text{bFilter}(\text{mtx} \ b_i, \text{mtx} \ b_o, \text{vec} \ p, \text{int} \ N) )</td>
</tr>
<tr>
<td>3</td>
<td>for ( k \leftarrow t_{id} ) to ( \text{num lines}(b_i) ) step ( W ) do</td>
</tr>
<tr>
<td>4</td>
<td>( l \leftarrow b_i[k]; )</td>
</tr>
<tr>
<td>5</td>
<td>if ( \text{memcmp}(l, p, N) == 0 ) then</td>
</tr>
<tr>
<td>6</td>
<td>( \text{memcpy}(l, b_o[k], N); )</td>
</tr>
</tbody>
</table>

A naive multi-thread implementation of \( \text{memcpy} \) 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 iterations 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 \( \text{memcpy} \) in a way that distributes operations on contiguous elements across consecutive threads. Algorithm 2 does it. Function \( \text{memcpy}_{\text{shfl}} \) is aware of the SIMD nature of a warp. Variables are stored as

---

3 The Intel SPMD Program Compiler (ISPC) is available at https://github.com/ispc/
vectors, having each position belonging to a specific thread. Instruction `shfl(v, i)` allows thread `tid` to read the value stored in variable `v`, but in the register space of thread `i`. This implementation gives 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`

```c
1 W ← warp size; tid ← thread index;
2 Function `memcpy_shfl`(vec s, vec d, int N)
3    for j ← 0 to W − 1 do
4        d[mj] ← shfl(d, j);
5        s[mj] ← shfl(s, j);
6        N[mj] ← shfl(N, j);
7    for i ← tid to Nmj − 1 step W do
8        d[i] ← s[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 `K2` from within a kernel `K1` [32]. In this case, programmers may request a large number of threads, i.e., multiple new warps in multiple thread blocks. As the inner `K2` is a new kernel, all its threads are active upon entry, regardless of branch divergence in `K1`. 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.

```c
1 W ← warp size; tid ← thread index;
2 Function `memcpy_dp`(vec s, vec d, int N)
3    for k ← tid 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.

Algorithm 4: SIMD Book Filter using CREV

```c
1 W ← warp size; tid ← thread index;
2 Function bFilter(mtx b, mtx h, vec p, int N)
3    for k ← tid to num_lines(b)-1 step W do
4        l ← b[k];
5        if `memcpy(l, p, N) = 0` then
6            `crev memcpy_crev(l, b_o[k], N);
7    `crev memcpy_crev(vec s, vec d, int N)
8    for k ← tid to N − 1 step W do
9        d[k] ← s[k];
```
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 \( \mu \text{-SIMD} \), a low-level programming language with a set of primitives that lets us implement CREV. In Section 3.3, we show how to implement the \texttt{crev} high-level construct using the building blocks available in \( \mu \text{-SIMD} \). 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 \texttt{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 \texttt{memcpy.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 \texttt{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, \textit{SIMD function call} and \textit{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, \( \mathcal{r} \)-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.

Data Distribution. Each formerly active warp thread is serialized to have a full warp operate on its data. As in the warp-synchronous \texttt{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, \( \mu \text{-SIMD} \). This language provides the low-
\[ \text{split}(\Theta, \beta, v) = (\Theta_0, \Theta_n) \text{ where} \]
\[ \Theta_0 = \{ t \mid t \in \Theta \text{ and } \beta[t] = \sigma_1 \text{ and } \sigma_1[v] = 0 \} \]
\[ \Theta_n = \{ t \mid t \in \Theta \text{ and } \beta[t] = \sigma_1 \text{ and } \sigma_1[v] \neq 0 \} \]

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

**Figure 3.** Auxiliary functions used to define \( \mu \)-SIMD. split is a filter, dividing threads into two divergent sets (\( \Theta_0 \) and \( \Theta_n \)). Auxiliary function push updates the synchronization stack \( \Pi \) due to control flow divergences.

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

To formalize the semantics of \( \mu \)-SIMD, we use the auxiliary functions shown in Figure 3. The semantics of \( \mu \)-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 \( (\Theta, \beta, \Sigma) \). The interface between Figure 4 and Figure 5 is performed by Rules BT and TL. The result of executing an arithmetic or logic instruction is a pair \( (\beta, \Sigma) \), i.e., they only update the program memory.

**The semantics of control flow divergences.** To simulate the effect of divergences, \( \mu \)-SIMD has a stack \( \Pi \). Each element in \( \Pi \) 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 \( \Pi \) 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. \( \Theta_{done} \) is the set of threads that have reached the synchronization point. \( \Theta_{todo} \) 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 \( \Lambda \) to \( \mu \)-SIMD. This stack is fundamental to the implementation of everywhere blocks. \( \Lambda \) holds pairs \( (\Theta, \Pi) \). Figure 4 shows that instructions everywhere (Rule EB) push elements onto \( \Lambda \), 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), Θall represents all the threads available in a warp. The thread stack lets us represent an unbounded number of different thread contexts; hence, programs might contain an 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 conditional jmp_mask. The statement jmp_mask(tid, l) will divert execution to l if tid is active in the mask at Λ’s top.

Example 3.2 Figure 6 shows a program written in µ-SIMD. We assume |Θall| = 4. This program increments a 4 × 4 matrix; however, line i is incremented only if Σ[i] = 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 tid = 0 and tid = 3 will enter the everywhere section, because Σ[0] = Σ[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 p1, . . . , T pn). 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
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.

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

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Var</th>
<th>Tid</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>v0 = _tid</td>
<td>v0</td>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>v1 = (v0 == 0)</td>
<td>v1</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>bz v1, Done</td>
<td></td>
<td>F F</td>
<td>1</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>v2 = v4 * (tid + 1)</td>
<td>v2</td>
<td></td>
<td>4</td>
<td>*</td>
<td>16</td>
<td>*</td>
</tr>
<tr>
<td>everywhere</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v8 = 0</td>
<td>v8</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Loop jmp_mask v8, Call jmp Next</td>
<td>v8</td>
<td></td>
<td>F F</td>
<td>F F</td>
<td>F F</td>
<td>F F</td>
</tr>
<tr>
<td>Call v3 = shfl(v2, v8)</td>
<td>v3</td>
<td></td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>v4 = v3 + tid</td>
<td>v4</td>
<td></td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>v5 = v4</td>
<td>v5</td>
<td></td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>v6 = v5 + 1</td>
<td>v6</td>
<td></td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>i1 = v6</td>
<td>i1</td>
<td></td>
<td>i1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Next v8 = v8 + 1</td>
<td>v8</td>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>v7 = (v8 == 4)</td>
<td>v7</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>bz v7, Loop</td>
<td></td>
<td>F F</td>
<td>F F</td>
<td>F F</td>
<td>F F</td>
<td>F F</td>
</tr>
<tr>
<td>Loop jmp_mask v8, Call jmp Next</td>
<td>v7</td>
<td></td>
<td>F F</td>
<td>F F</td>
<td>F F</td>
<td>F F</td>
</tr>
<tr>
<td>Next v8 = v8 + 1</td>
<td>v8</td>
<td></td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>v7 = (v8 == 4)</td>
<td>v7</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>bz v7, Loop</td>
<td></td>
<td>F F</td>
<td>F F</td>
<td>F F</td>
<td>F F</td>
<td>F F</td>
</tr>
<tr>
<td>Loop jmp_mask v8, Call jmp Next</td>
<td>v8</td>
<td></td>
<td>F F</td>
<td>F F</td>
<td>F F</td>
<td>F F</td>
</tr>
<tr>
<td>Call v3 = shfl(v2, v8)</td>
<td>v3</td>
<td></td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>v4 = v3 + tid</td>
<td>v4</td>
<td></td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>v5 = v4</td>
<td>v5</td>
<td></td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>v6 = v5 + 1</td>
<td>v6</td>
<td></td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>i1 = v6</td>
<td>i1</td>
<td></td>
<td>i1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Next v8 = v8 + 1</td>
<td>v8</td>
<td></td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>v7 = (v8 == 4)</td>
<td>v7</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>bz v7, Loop</td>
<td></td>
<td>F F</td>
<td>F F</td>
<td>F F</td>
<td>F F</td>
<td>F F</td>
</tr>
<tr>
<td>every</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if i = 1</td>
<td></td>
<td></td>
<td></td>
<td>F F</td>
<td>F F</td>
<td>F F</td>
</tr>
<tr>
<td>done</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 9. A program written in ISPC, and the tree showing function calls for T0.

Algorithm 5: Data distribution

1 Function declaration: f(p1, ... , pn);
2 Function call: f(t1, a1, ... , tn, an);
3 Function extract(tn, p, a, 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);

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., \( \Theta_\text{all} = \{T0, T1, T2, T3\} \). When foo is in-
voked, 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 $\mu$-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 $\mu$-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 $\Lambda$ 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 composability, we have that, in the absence of divergences, a sequence of $n$ nested crev calls will create $|\Theta_{at}|^N$ tasks. Notice that CREV produces new tasks, but not new threads: we still have only $|\Theta_{at}|$ threads to solve these tasks.

**Commutativity.** The translator of Figure 8 calls an r-function in a lexicographic order defined by thread iden-
tifiers. However, $\mu$-SIMD’s primitives do not impose any order on the threads pushed onto $\Lambda$. 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 $\Pi$, 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, $\mu$-SIMD’s everywhere instruction pushes onto $\Lambda$, together with the set of active threads, the divergent state $\Pi$.

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 $|\Theta_{at}|^N$ 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 $|\Theta_{at}| = 4$. Thus, three non-divergent nested r-calls will create $4 \times 4 \times 4 = 256$ 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.

<table>
<thead>
<tr>
<th>Program</th>
<th>Thread Stack</th>
<th>Activation Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>main() {</td>
<td>$T_0$: foo</td>
<td>main:</td>
</tr>
<tr>
<td>var int a = $T_0$;</td>
<td>$T_1$: foo</td>
<td>$T_1$:</td>
</tr>
<tr>
<td>crev foo(a);</td>
<td>$T_2$: foo</td>
<td>$T_2$:</td>
</tr>
<tr>
<td>bar(b)</td>
<td>$T_3$: foo</td>
<td>$T_3$:</td>
</tr>
<tr>
<td></td>
<td>$T_0$: foo, $T_0$: bar</td>
<td></td>
</tr>
<tr>
<td>foo(uni int a) {</td>
<td>$T_1$: foo</td>
<td></td>
</tr>
<tr>
<td>var int b = $a + T_0$;</td>
<td>$T_1$: bar</td>
<td></td>
</tr>
<tr>
<td>crev bar(b) }</td>
<td>$T_2$: foo, $T_2$: bar</td>
<td></td>
</tr>
<tr>
<td>bar(uni int b) {</td>
<td>$T_3$: foo, $T_3$: bar</td>
<td></td>
</tr>
<tr>
<td>uni int c = b + 1;</td>
<td>$T_3$: foo, $T_3$: bar</td>
<td></td>
</tr>
<tr>
<td>crev baz(c) }</td>
<td>$T_3$: foo, $T_3$: bar</td>
<td></td>
</tr>
<tr>
<td>baz(uni int c) {</td>
<td>$T_3$: foo, $T_3$: bar</td>
<td></td>
</tr>
<tr>
<td>var int d = c + 1;</td>
<td>$T_3$: foo, $T_3$: bar</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$T_3$: foo, $T_3$: bar</td>
<td></td>
</tr>
</tbody>
</table>

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

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

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.
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 version 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. \( P \leftarrow \text{pattern}; T \leftarrow \text{target text}; \)
2. \( W \leftarrow \text{warp size}; t_{id} \leftarrow \text{thread index}; \)
3. Function \( \text{memcmp(Offset } k) \)
4. \( m \leftarrow \text{True}; \)
5. \( \text{for } i \leftarrow t_{id} \text{ to } |P| \text{ do } \)
6. \[ \text{if } P[i] \neq T[i+k] \text{ then } m \leftarrow \text{False}; \]
7. \( \text{if all } (m = \text{True}) \text{ then } \text{Found(k)}; \)
8. Function \( \text{StringMatch} \)
9. \( \text{for } i \leftarrow t_{id} \text{ to } (|T| - |P|) \text{ step } W \text{ do } \)
10. \[ \text{if } P[0] = T[i] \text{ then } \text{crev memcmp(i)}; \]
11. Function \( \text{ParStringMatch} \)
12. \( \text{for } i \leftarrow t_{id} \text{ to } (|T| - |P|) \text{ step } W \text{ do } \)
13. \( j \leftarrow 0; k \leftarrow i; \)
14. \( \text{while } j < |P| \text{ and } P[j] = T[k] \text{ do } \)
15. \( j \leftarrow j + 1; k \leftarrow k + 1; \)
16. \( \text{if } j = |P| \text{ then } \text{Found(k)}; \)

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 \( \text{memcmp} \) 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, . . . , 28, 32 in Jane Austen’s book Pride and Prejudice, taken from Project Gutenberg. 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 \( \text{memcmp} \) 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.

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 threads 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 gen-

---

4 Function \( \text{memcmp} \) is also used at line 5 of Algorithm 1

5 [https://www.gutenberg.org/ebooks/42671](https://www.gutenberg.org/ebooks/42671)
eral 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 parallelism. In other words, it changes the loop structure by distributing iterations across different threads. We are not competing with these efforts, because CREV is not an alternative to dynamic parallelism; hence, it does not create extra 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.

**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

**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. 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/](http://cuda.dcc.ufmg.br/swan/).

---

6 See [https://nvlabs.github.io/cub/](https://nvlabs.github.io/cub/)
References


A. Artifact Evaluation

This artifact consists of a virtual machine, in which we have installed the required software for reproducing our experiments. The file has 9.2 GB uncompressed, 1.6 GB compressed, and is available for download at http://cuda.dcc.ufmg.br/swan/. In that webpage we have charts for some experiments that we could not fit in the original paper. We also have a Prolog interpreter for the semantics of μ-SIMD, at http://cuda.dcc.ufmg.br/swan/src/semanitics.zip. The artifact itself can be downloaded through the link: http://cuda.dcc.ufmg.br/swan/artifact.tar.gz.

Hardware Dependencies  The hardware requirements are:
- host with x86 64bit processor;
- minimum of 30GB of free space;
- have at least 8 cores available for the virtual machine.

Software Dependencies  Our work is entirely implemented on top of Intel's Single Program Multiple Data compiler, namely ISPC. We thus depend on its source code to apply our patch, and thereafter enable our crev idiom. ISPC, in its turn, is implemented on top of LLVM, and uses Bison and Flex as parsing tool. Those sum up as all software dependences for our work.

ISPC for Linux 64-bit is available for download at sourceforge: http://ispc.github.io/downloads.html (version 1.9.1). Bison may be checked-out straight from the linux distro repository, using the command sudo apt-get install bison. Notice that the version currently installed is GNU Bison 3.0.2, and a version ≥ 2.4 is recommended. If not available from repository, please, try and download it from https://ftp.gnu.org/gnu/bison/. Flex may be found at http://flex.sourceforge.net/ and the version should be ≥ 2.5 (currently installed in the VM is version 2.5.39), and can be also checked-out from linux repository using the command: sudo apt-get install flex. And as for LLVM, this well-known compiler architecture is available online at http://llvm.org (version 3.7.1).

Data sets  To validate our approach, we have executed our analysis on our hand-crafted set of benchmarks, namely CREV-ISPC. We targeted as many different applications as possible, ranging from string processing and graph searches to sort algorithms. Our benchmarks are available alongside with this artifact, in the directory /home/crev/benchmarks.

A.1 Installation

Our artifact is provided as a virtual machine image. To use it, you need to download and install the VirtualBox VM player7. To execute the virtual machine, do the following:
- download and uncompress the VM image;
- open VirtualBox;
- create a new virtual machine, by clicking in Machine → New...
- give a name to the new VM, by filling the Name field;
- in Type, select Linux;
- in Version, select Ubuntu (64 bit);
- click continue, select the desired amount of RAM memory, and press continue again;
- in the hard drive selection screen, select the option Use an existing virtual hard drive file, then select the .vdi file containing the VM image that you just downloaded;
- your virtual machine is ready to use! To start, simply double-click it in the VirtualBox VM list.

At startup, you will be prompted for a username and password. Use the following combination:

```plaintext
login: crev
password: crev@ppopp17
```

After logging in, a command prompt will appear and the VM is ready to use. Even though the VM image has a command line ready to use, we highly recommend the use of SSH port forwarding, so the user can simply SSH into the Virtual Machine to execute commands. The VM has Ubuntu 15.04 installed on it. There is no graphic interface, so all actions are executed through command line.

A.1.1 LLVM

LLVM is a well-known compiler architecture. In this work, we use LLVM-3.7.1. To proceed with its installation, do the following:
- Download llvm-3.7.1: codes/s4.3.2-1.sh
- Download cfe-3.7.1: codes/s4.3.2-2.sh
- Extract llvm and cfe: codes/s4.3.2-3.sh
- Move cfe folder to the path llvm/tools/clang:
codes/s4.3.2-4.sh
- Create a build directory for llvm, configure, and install it: codes/s4.3.2-5.sh

A.1.2 Bison and Flex

Apart from LLVM, ISPC has two other dependences: Bison, a YACC-compatible parser generator; and Flex, a fast lexical analyzer generator. Check if you already have the dependences solved, and otherwise install the required software. Use the following command lines to install from your linux distro repository and check versions installed. Bison must be ≥ 2.4, and Flex must be ≥ 2.5. codes/s4.3.3-1.sh

A.1.3 CREV-ISPC

Before moving ahead with the installation of ISPC, set the path to your LLVM build, so ISPC will be able to use it by
default. Add the following lines to your `/home/crev/.bashrc` file. codes/s4.4-1.sh

There is a quirk about the dependency with LLVM, which consists of a missing file. It may be related to the version of LLVM being used in our implementation (3.7.1), but it is necessary to copy the file `DiagnosticCommonKinds.inc`, generated after LLVM/Clang build, to the source directory of LLVM. The command below should solve the issue.

```
codes/s4.4-2.sh
```

Now that all dependences have been set up, download our CREV-extended version of Intel’s SPMD compiler (ISPC). It is available at [http://cuda.dcc.ufmg.br/swan/src/crev-ispc.tar.gz](http://cuda.dcc.ufmg.br/swan/src/crev-ispc.tar.gz). Once downloaded, extract it and install CREV-ISPC. codes/s4.4-3.sh

ISPC’s Makefile comes originally configured to automatically fetch the path to LLVM build – as long as it has been informed in variable `$PATH`]. However, in case you step into path problems, set the variables to your needs, as you shall find set in our current version of ISPC’s Makefile. The variables below show our specific paths within ispc/Makefile. [linenos, firstnumber=65]codes/s4.4-4.sh

```bash

A.2 Workflow

To validate our approach, we have executed our analysis on our hand-crafted set of benchmarks, namely CREV-ISPC. Our goal was to target different scenarios in which it may be useful to call SIMD-only function within divergent regions. The selected set of applications implemented is a spread consisting of string matching, graph traversals and sort algorithms. As stated in our paper, our approach provides a facility that helps programmers write efficient code without entangling oneself in complex synchronization-dependent structures. Our benchmarks are available alongside with this artifact, and can be found in the directory `/home/crev/benchmark`.

A.2.1 Runtime Environment

In the core paper, 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 gave us warps with eight threads, e.g., $|\theta_{\text{all}}| = 8$. In the case of our artifact, we have enabled 4 cores in a computer with two 4-core 2.60GHz Intel i7-3720QM CPU, also with 8-wide AVX vector units, but running Linux Ubuntu 15.04.

A.2.2 Benchmarks

We provide seven benchmarks: (1) Book Filter; (2) String Matching; (3) Bellman-Ford; (4) Depth-First Search; (5) Leader Election; (6) Quick-Sort and (7) Merge-Sort. The numbers in parentheses are used in Section A.2.3 to identify the input data-set of each benchmark. To build up the binaries, simply enter the selected benchmark directory and run `make`. Each benchmark varies, be it in the number of binary files generated, output and input format, and input dataset. To this end, we leave, within each folder, a bash script named `run.sh`, which executes the benchmarks. We also have a parser for each of the experiments, in order to extract and list the results in a human-friendly fashion. The parsing script is also a shell script, named `parse.sh`. While compiling, the makefile may raise some warnings, but there is no actual error.

A.2.3 Input Datasets

The table below relates the datasets we have used in the experiments and the associated benchmarks that received the data as input. The datasets are all available in the folder `/home/crev/datasets`. Users of our artifact can produce different inputs to carry out more extensive experiments.

<table>
<thead>
<tr>
<th>Users</th>
<th>Dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Binary strings</td>
</tr>
<tr>
<td></td>
<td>10K strings of 0s and 1s, each with 20K bits, and random target pattern with 16 bits.</td>
</tr>
<tr>
<td>2</td>
<td>Text books</td>
</tr>
<tr>
<td></td>
<td>256MB of text taken from books available in the Project Gutenberg; target pattern is 16 chars.</td>
</tr>
<tr>
<td>3</td>
<td>Erdős-Rényi</td>
</tr>
<tr>
<td></td>
<td>random Erdős-Rényi graph with 2,048 nodes and 80% edge probability.</td>
</tr>
<tr>
<td>4, 5</td>
<td>Full octree</td>
</tr>
<tr>
<td></td>
<td>8-ary complete tree with depth up to 5, i.e., root + five full levels.</td>
</tr>
<tr>
<td>6, 7</td>
<td>Random integers</td>
</tr>
<tr>
<td></td>
<td>16K random integers, ranging from [0, 100000).</td>
</tr>
</tbody>
</table>

A.3 Evaluation and Expected Result

During evaluation of the artifact, one may either relate to the paper, in order to validate the results therein presented; or analyze the outputs w.r.t. different implementations: compare execution times for sequential, parallel, crev-based, and launch-based implementations. Either way, the experiments should also take into account the main contribution of the paper: we propose an idiom that combines the simplicity of dynamic parallelism, with the efficiency of warp-synchronous programming. Therefore, as presented in the paper, the results tend to bring up speedups, but this is not expected to hold true for all cases: execution time may vary significantly according to input and algorithm specific parameters. Further information is available online, including graphs with results obtained in the original runtime environment – in which the experiments in the paper were executed. The link to our webpage is [http://cuda.dcc.ufmg.br/swan](http://cuda.dcc.ufmg.br/swan).

B. Reviewing Methodology

This artifact has been reviewed according to the guidelines established by the Artifact Evaluation Committee of CGO and PPoPP. This methodology is described at [http://cTuning.org/ae/submission-20161020.html](http://cTuning.org/ae/submission-20161020.html)