Function Call Re-vectorization

Authors: Rubens E. A. Moreira, Sylvain Collange, Fernando M. Q. Pereira
Our goal is to increase the **programmability** of languages that target **SIMD-like** machines, without sacrificing efficiency.

Programmability of algorithms involving function calls:
- Quicksort
- Depth-First Search
- Leader election
- String matching
- etc

SIMD-like hardware
- GPUs
- SSE vector units
- AVX vector units
- etc

SIMD inside SIMD: we vectorize inner loops inside a vectorized outer loop
Example: string matching

```c
void str_compare(int offset) {
    bool m = true;
    for (int i=threadId.x; i < |P|; i+=threadDim.x)
        if (P[i] != T[i + offset]) { m = false; break; }
    if (all(m == true)) Found();
}

void StringMatch() {
    for (int i=threadId.x; i < (|T| - |P|); i+=threadDim.x)
        if (P[0] == T[i]) crev str_compare(i);
}
```

Example: if the warp size is 32 and 7 threads are enabled when the program flow hits this line, all 32 threads execute `str_compare` 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.
DIVERGENCES
Divergences

```c
void kernel(int **A, int **B, int N) {
    int tid(threadId.x);
    if (tid < 3) {
        memcpy(A[tid], B[tid], N);
    } else {
        ;
    }
}
```

Kernel for parallel execution (CUDA).
Divergences

```c
void kernel(int **A, int **B, int N) {
    int tid(threadId.x);
    if (tid < 3) {
        memcpy(A[tid], B[tid], N);
    } else {
        ;
    }
}
```

Control flow graph for kernel.

Kernel for parallel execution (CUDA).
Divergences

void kernel(int **A, int **B, int N) {
    int tid(threadId.x);
    if (tid < 3) {
        memcpy(A[tid], B[tid], N);
    } else {
        ;
    }
}

Kernel for parallel execution (CUDA).

SIMD: LOCKSTEP EXECUTION!

Control flow graph for kernel.
void kernel(int **A, int **B, int N) {
    int tid(threadId.x);
    if (tid < 3) {
        memcpy(A[tid], B[tid], N);
    } else {
        ;
    }
}

Kernel for parallel execution (CUDA).

SIMD: LOCKSTEP EXECUTION!

if (threadId.x < 3)

then
    memcpy(A, B, N);

else

DIVERGENCE!

Control flow graph for kernel.
```c
void kernel(int **A, int **B, int N) {
    int tid(threadId.x);
    if (tid < 3) {
        memcpy(A[tid], B[tid], N);
    } else {
        ;
    }
}
```

Kernel for parallel execution (CUDA).

**SIMD: LOCKSTEP EXECUTION!**

Control flow graph for kernel.
void kernel(int **A, int **B, int N) {
    int tid(threadId.x);
    if (tid < 3) {
        memcpy(A[tid], B[tid], N);
    } else {
        ;
    }
}

Kernel for parallel execution (CUDA).

SIMD: LOCKSTEP EXECUTION!

if (threadId.x < 3)

then
    memcpy(A, B, N);

else

Control flow graph for kernel.
void kernel(int **A, int **B, int N) {
    int tid(threadId.x);
    if (tid < 3) {
        memcpy(A[tid], B[tid], N);
    } else {
        ;
    }
}

Kernel for parallel execution (CUDA).

Divergent region: only active threads run memcpy

Control flow graph for memcpy.
Divergence: CUDA

```c
void kernel(int **A, int **B, int N) {
    int tid(threadId.x);
    if (tid < 3) {
        memcpy(A[tid], B[tid], N);
    } else {
        //
    }
}
```

Kernel for parallel execution (CUDA).

Divergent region: only active threads run `memcpy`
Divergences: Cuda

void kernel(int **A, int **B, int N) {
    int tid(threadId.x);
    if (tid < 3) {
        memcpy(A[tid], B[tid], N);
    } else {
    }
}

Kernel for parallel execution (CUDA).

Divergent region: only active threads run memcpy

Control flow graph for memcpy.
Divergences: Cuda

Kernel for parallel execution (CUDA).

Divergent region: only active threads runMemcpy.

Control flow graph formemcpy.

observed behavior:

void kernel(int **A, int **B, int N) {
    int tid(threadId.x);
    if (tid < 3) {
        memcpy(A[tid], B[tid], N);
    } else {
        ;
    }
}

Suboptimal behavior: thread $T_3$ is not active and memory accesses are scattered.

Control flow graph for memcpy.

$T_0$ $T_1$ $T_2$ $T_3$
DYNAMIC PARALLELISM
Dynamic Parallelism

```c
void kernel(int **A, int **B, int N) {
    int tid(threadId.x);
    if (tid < 3) {
        memcpy<<<1, 4>>>(A[tid], B[tid], N);
    } else {
        ;
    }
}
```

Kernel for parallel execution (CUDA).

CUDA’s special syntax for **dynamic parallelism**: `kernel<<<#blocks, #threads_per_block>>>(args...)`
Dynamic Parallelism

```c
void memcpy(int *dest, int *src, int N) {
    for (int i=threadId.x; i < N; i+=threadDim.x) {
        dest[i] = src[i];
    }
}
```

SIMD implementation of memory copy.

Dynamic parallelism changes the **dimension** of the parallelism

Actual behavior with CUDA’s dynamic parallelism:

```plaintext
<table>
<thead>
<tr>
<th>Threads</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0</td>
<td>a0</td>
</tr>
<tr>
<td>t1</td>
<td>b0</td>
</tr>
<tr>
<td>t2</td>
<td>c0</td>
</tr>
<tr>
<td>t3</td>
<td>d0</td>
</tr>
</tbody>
</table>
```

From T0:
- T0: a0, a1, a2, a3, a4
- T1: b0, b1, b2, b3, b4
- T2: c0, c1, c2, c3, c4

From T1:
- T0: a0, a1, a2, a3, a4
- T1: b0, b1, b2, b3, b4
- T2: c0, c1, c2, c3, c4

From T2:
- T0: a0, a1, a2, a3, a4
- T1: b0, b1, b2, b3, b4
- T2: c0, c1, c2, c3, c4

**memcpy** runs once per active thread at `memcpy<<1,4>>` call site!
Dynamic Parallelism

```c
void kernel(int **A, int **B, int N) {
    int tid(threadId.x);
    if (tid < 3) {
        memcpy<<<1, 4>>>(A[tid], B[tid], N);
    } else {
        ;
    }
}
```

Kernel for parallel execution (CUDA).

kernel<<<#blocks, #threads>>>(args...);

CUDA’s Dynamic Parallelism:
- Nested kernel calls

Has the **overhead** of allocating and scheduling a new kernel

Parallel Time = Kernel Launching Overhead + Sequential Time

#blocks x #threads
FUNCTION CALL RE-VECTORIZATION
Looking at Dynamic Parallelism in a New Way

- **Don't create new threads**

- **Reuse threads**
  - Save the state of threads in flight
  - Give them new work
  - Restore their state once they are done

- **Everywhere blocks** (early languages for SIMD machines)
  - C*
  - MPL
  - POMPC

- **Shuffle** (warp aware instruction)
  - `shfl(v,i)` allows thread to read the value stored in variable `v`, but in the register space of thread `i`
Warp-Synchronous Programming: Everywhere blocks

```c
void memcpy(int *dest, int *src, int N) {
    for (int i=threadId.x; i < N; i+=threadDim.x) {
        dest[i] = src[i];
    }
}
```

SIMD implementation of memory copy.

**Everywhere blocks:**
Warp-Synchronous Programming: Everywhere blocks

```c
void memcpy(int *dest, int *src, int N) {
    for (int i=threadId.x; i < N; i+=threadDim.x) {
        dest[i] = src[i];
    }
}
```

SIMD implementation of memory copy.

Everywhere blocks:

```
DIVERGENCE:  T_0  T_1  T_2  T_3
```

Warp-Synchronous Programming: Everywhere blocks

```c
void memcpy(int *dest, int *src, int N) {
    for (int i=threadId.x; i < N; i+=threadDim.x) {
        dest[i] = src[i];
    }
}
```

SIMD implementation of memory copy.

**Everywhere blocks:**

- **DIVERGENCE:**
  - T₀
  - T₁
  - T₂
  - T₃

- **EVERYWHERE:**
  - T₀
  - T₁
  - T₂
  - T₃

All threads are **temporarily re-enabled** to process code within EVERYWHERE block!
Warp-Synchronous Programming: Everywhere blocks

```c
void memcpy(int *dest, int *src, int N) {
    for (int i=threadId.x; i < N; i+=threadDim.x) {
        dest[i] = src[i];
    }
}
```

SIMD implementation of memory copy.

**Everywhere blocks:**

- **DIVERGENCE:**
  - T₀
  - T₁
  - T₂
  - T₃

- **EVERYWHERE:**
  - T₀
  - T₁
  - T₂
  - T₃

  ```c
  EVERYWHERE {
  code...
  }
  ```

- **DIVERGENCE:**
  - T₀
  - T₁
  - T₂
  - T₃

All threads are temporarily re-enabled to process code within EVERYWHERE block!

Divergences restored!
void memcpy(int *dest, int *src, int N) {
    for (int i=threadId.x; i < N; i+=threadDim.x) {
        dest[i] = src[i];
    }
}

SIMD implementation of memory copy.

void memcpywrapper(int **dest, int **src, int N, int mask) {
    EVERYWHERE {
        for (int i=0; i < threadDim.x; ++i) {
            if (not (mask & (1 << i))) continue; // skip thread “i”
            dest_i = shuffle(dest, i);           // if it is divergent
            src_i = shuffle(src, i);
            N_i = shuffle(N, i);
            memcpy(dest_i, src_i, N_i);
        }
    }
}

Low-level wrapper for SIMD memory copy.

memcpy is called once per active thread, but every call is executed by every thread.
Warp-Synchronous Programming: Everywhere blocks

```c
void memcpy(int *dest, int *src, int N) {
    for (int i=threadId.x; i < N; i+=threadDim.x) {
        dest[i] = src[i];
    }
}
```

SIMD implementation of memory copy.

```c
void memcpy_wrapper(int **dest, int **src, int N, int mask) {
    EVERYWHERE {
        for (int i=0; i < threadDim.x; ++i) {
            if (not (mask & (1 << i))) continue; // skip thread "i"
            dest_i = shuffle(dest, i);  // if it is divergent
            src_i = shuffle(src, i);
            N_i = shuffle(N, i);
            memcpy(dest_i, src_i, N_i);
        }
    }
}
```

Low-level wrapper for SIMD memory copy.

1. *everywhere* re-enables all threads!
Warp-Synchronous Programming: Everywhere blocks

```c
void memcpy(int *dest, int *src, int N) {
    for (int i=threadId.x; i < N; i+=threadDim.x) {
        dest[i] = src[i];
    }
}
```

SIMD implementation of memcpy:

```c
void memcpy_wrapper(int ***dest, int ***src, int N, int mask) {
    EVERYWHERE {
        for (int i=0; i < threadDim.x; ++i) {
            if (not (mask & (1 << i))) continue; // skip thread “i”
            dest_i = shuffle(dest, i);           // if it is divergent
            src_i = shuffle(src, i);
            N_i  = shuffle(N, i);
            memcpy(dest_i, src_i, N_i);
        }
    }
}
```

Low-level wrapper for SIMD memory copy.

1. **everywhere** re-enables all threads!
2. Skip formerly divergent threads!

Warp-Synchronous Programming: Everywhere blocks
Warp-Synchronous Programming: Everywhere blocks

void memcpy(int *dest, int *src, int N) {
    for (int i=threadId.x; i < N; i+=threadDim.x) {
        dest[i] = src[i];
    }
}

SIMD implementation of memcpy:

void memcpy_wrapper(int **dest, int **src, int N, int mask) {
    EVERYWHERE {
        for (int i=0; i < threadDim.x; i++) {
            if (not (mask & (1 << i))) continue; // skip thread "i"
            dest_i = shuffle(dest, i);           // if it is divergent
            src_i = shuffle(src, i);
            N_i = shuffle(N, i);
            memcpy(dest_i, src_i, N_i);
        }
    }
}

Low-level wrapper for SIMD memory copy.

1. everywhere re-enables all threads!
2. Skip formerly divergent threads!
3. Extracts values for current thread “i”.
void memcpy(int *dest, int *src, int N) {
  for (int i=threadId.x; i < N; i+=threadDim.x) {
    dest[i] = src[i];
  }
}

SIMD implementation of memory copy:

void memcpy_wrapper(int **dest, int **src, int N, int mask) {
  EVERYWHERE {
    for (int i=0; i < threadDim.x; ++i) {
      if (not (mask & (1 << i))) continue; // skip thread “i”
      dest_i = shuffle(dest, i);           // if it is divergent
      src_i = shuffle(src, i);
      N_i  = shuffle(N, i);
      memcpy(dest_i, src_i, N_i);
    }
  }
}

1. **everywhere** re-enables all threads!
2. Skip formerly divergent threads!
3. Extracts values for current thread “i”.
4. We then call our SIMD kernel **memcpy**.

Low-level wrapper for SIMD memory copy.
Function Call Re-Vectorization: CREV

```c
void memcpy(int *dest, int *src, int N) {
    for (int i[threadId.x; i < N; i+=threadDim.x) {
        dest[i] = src[i];
    }
}
```

SIMD implementation of memory copy.

```c
void memcpy_wrapper(int **dest, int **src, int N, int mask) {
    crev memcpy(dest, src, N);
}
```

High-level view

**Example:** if the warp size is 32 and 7 threads are enabled when the program flow hits this line, all 32 threads execute `str_compare` 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.
Warp-Synchronous Programming: Everywhere blocks

We have defined the low-level building blocks of crev:

\[
\begin{align*}
P[pc] &= \text{stop} \quad \frac{}{(\Theta, \beta, \Sigma, \emptyset, \Lambda, P, pc) \rightarrow (\Theta, \beta, \Sigma)} \\
(\text{Sp})

P[pc] &= \text{bz } v, l \quad \text{split}(\Theta, \beta, v) = (\Theta, \emptyset) \\
push(\Pi, \emptyset, pc, l) &= \Pi' \quad (\Theta, \beta, \Sigma, \Pi', \Lambda, P, p) \rightarrow (\Theta', \beta', \Sigma') \\
(\text{Br})

P[pc] &= \text{bz } v, l \quad \text{split}(\Theta, \beta, v) = (\emptyset, \Theta) \\
push(\Pi, \emptyset, pc, l) &= \Pi' \quad (\Theta, \beta, \Sigma, \Pi', \Lambda, P, p + 1) \rightarrow (\Theta', \beta', \Sigma') \\
(\text{Br})

P[pc] &= \text{branch_mask } T_{id}, l \\
T_{id} \in \Theta' \quad (\Theta, \beta, \Sigma, \Pi, (\Theta', \Pi') : \Lambda, P, l) \rightarrow (\Theta'', \beta'', \Sigma'') \\
(\text{Ba})

P[pc] &= \text{branch_mask } T_{id}, l \\
T_{id} \notin \Theta' \quad (\Theta, \beta, \Sigma, \Pi, (\Theta', \Pi') : \Lambda, P, l + 1) \rightarrow (\Theta'', \beta'', \Sigma'') \\
(\text{Ba})

\end{align*}
\]

\[
\begin{align*}
\text{Semantics of } \text{everywhere} \text{ in SIMD:} \\
\text{encode the building blocks to implement this construct}
\end{align*}
\]
We have defined the low-level building blocks of crev:

Implemented an **abstract SIMD machine** in Prolog, with support to **everywhere** blocks.

Extended Intel's SPMD compiler with a **new idiom, function call re-vectorization**, that enhances native dynamic parallelism.
Function Call Re-Vectorization: CREV

crev memcmp(i)

---

Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>v0 = ↓tid</td>
<td></td>
</tr>
<tr>
<td>v1 = (v0 == 0)</td>
<td></td>
</tr>
<tr>
<td>bz v1, Done</td>
<td></td>
</tr>
<tr>
<td>v2 = 4 * (tid + 1)</td>
<td></td>
</tr>
<tr>
<td>everywhere</td>
<td></td>
</tr>
<tr>
<td>v8 = 0</td>
<td></td>
</tr>
<tr>
<td>Loop</td>
<td></td>
</tr>
<tr>
<td>jmp_mask v8, Call</td>
<td></td>
</tr>
<tr>
<td>Next</td>
<td></td>
</tr>
<tr>
<td>v3 = shfl(v2, v8)</td>
<td></td>
</tr>
<tr>
<td>v4 = v3 + tid</td>
<td></td>
</tr>
<tr>
<td>v5 = ↓v4</td>
<td></td>
</tr>
<tr>
<td>v6 = v5 + 1</td>
<td></td>
</tr>
<tr>
<td>↑v4 = v6</td>
<td></td>
</tr>
<tr>
<td>v8 = v8 + 1</td>
<td></td>
</tr>
<tr>
<td>v7 = (v8 == 4)</td>
<td></td>
</tr>
<tr>
<td>bz v7, Loop</td>
<td></td>
</tr>
<tr>
<td>end_everywhere</td>
<td></td>
</tr>
<tr>
<td>↑tid = 1</td>
<td></td>
</tr>
<tr>
<td>Done</td>
<td></td>
</tr>
<tr>
<td>sync</td>
<td></td>
</tr>
</tbody>
</table>

---

Function declaration: \( f(p_1, \ldots, p_n); \)
Function call: \( f(t_1 a_1, \ldots, t_n a_n); \)
Function extract \( (t^n; p^n, a^n, i); \)

for \( k \in 1 \ldots n \) do
  if \( t_k == \text{uniform} \) then
    \( p_k = a_k; \)
  if \( t_k == \text{varying} \) then
    \( \text{shfl}(a_k, i); \)
end

---

1. everywhere ; begin CREV
2. \( i = 0 \) ; Loop counter
3. loop : jmp_mask i, call ; Skip idle threads
4. jmp next ; function call
5. call : extract \( (t^n; p^n, a^n, i) \) ; Algorithm 5
6. “call” \( f \)
7. next : \( i = i + 1 \)
8. \( \text{bnz}(i \neq W) \) loop
9. end_everywhere ; end CREV
Properties of CREV:

- **Composability**
  We are able to nest everywhere blocks: crev can be called recursively!

- **Multiplicative composition**
  The target crev function runs once per active thread.
  In a warp of $W$ threads, the function may run up to $W$ times.
  If the call is recursive, up to $W^N$ times.

- **Commutativity**
  There is no predefined order between execution of crev’s target function.
EVALUATION
Experimental Setup

- **ISPC**, the Intel SPMD Program Compiler, v 1.9.1
  - Compiler implemented on top of LLVM
  - Programming language (extension of C)

- **Benchmarks**: seven algorithms implemented in different ways:
  - **CREV**: our contribution
  - **PAR**: implementation based on ISPC's constructs
  - **SEQ**: state-of-the-art sequential implementation
  - **Launch**: implementation using dynamic parallelism (pthreads)

- **Environment**
  - 6-core 2.00 GHz Intel Xeon E5-2620 CPU with 8-wide AVX vector units
  - Linux Ubuntu 12.04 3.2.0
The Competitors

**CREV and Launch**: implementation using function calls:

```c
void memcmp(int offset) {
    bool m = true;
    for (int i=threadId.x; i < |P|; ++i)
        if (P[i] != T[i + offset]) m = false;
    if (all(m == true)) Found(k);
}
void StringMatch() {
    for (int i=threadId.x; i < (|T| - |P|); i+=threadDim.x)
        if (P[0] == T[i]) (crev/launch) memcmp(i);
}
```

**PAR**: implementation using constructs available in ISPC

```c
void ISPC_StringMatch() {
    for (int i=threadId.x; i < (|T| - |P|); i+=threadDim.x) {
        int j = 0, k = i;
        while (j < |P| and P[j] == T[k])
            j = j + 1; k = k + 1;
        if (j == |P|) Found(k);
    }
}
```

**SEQ**: state-of-the-art algorithm.
Example: Knuth-Morris-Pratt pattern matching algorithm.
Numbers show percentage of speedup of CREV over PAR (white) and KMP (Grey)

Input: 256MB in 5M lines from books from Project Gutenberg
Leader-Election

Numbers show percentage of speedup of CREV over PAR (white), SEQ (Grey) and Launch (black).

**Input:** 8-ary complete tree of depth 5 (root + five full levels of nodes)
# Function Call Re-Vectorization: CREV

## Execution times (in millions of cycles):

<table>
<thead>
<tr>
<th></th>
<th>Sequential</th>
<th>Parallel</th>
<th>Launch</th>
<th>CREV</th>
<th>Dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>BookFilter</td>
<td>--</td>
<td>8530.990</td>
<td>7857.980</td>
<td>7405.175</td>
<td>bin-L20K-P16</td>
</tr>
<tr>
<td></td>
<td>not implemented</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>String Matching</td>
<td>6649.279</td>
<td>3576.143</td>
<td>393166.268</td>
<td>2737.939</td>
<td>txt-256MB-P16</td>
</tr>
<tr>
<td></td>
<td>KMP Algorithm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bellman-Ford</td>
<td>141088.730</td>
<td>493619.688</td>
<td>--</td>
<td>529856.065</td>
<td>erdos-renyi</td>
</tr>
<tr>
<td></td>
<td>not implemented</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Depth-First Search</td>
<td>3754.101</td>
<td>3786.263</td>
<td>--</td>
<td>3790.444</td>
<td>octree-D5</td>
</tr>
<tr>
<td></td>
<td>not implemented</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Connected-Component Leader</td>
<td>4054.658</td>
<td>3983.088</td>
<td>5272.919</td>
<td>3984.795</td>
<td>octree-D5</td>
</tr>
<tr>
<td>Quicksort-bitonic</td>
<td>2.871</td>
<td>--</td>
<td>204.278</td>
<td>2.878</td>
<td>int-16K</td>
</tr>
<tr>
<td></td>
<td>not implemented</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mergesort-bitonic</td>
<td>7.302</td>
<td>--</td>
<td>104.985</td>
<td>4.114</td>
<td>int-16K</td>
</tr>
<tr>
<td></td>
<td>not implemented</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Datasets:**

- **bin-L20K-P16**: 10K strings of 0s and 1s, each of length 20K, and target pattern of length 16.
- **txt-256MB-P16**: 256MB in 5M lines from books from Project Gutenberg; target pattern has length 16.
- **erdos-renyi**: random Erdos-Renyi graph with 2048 nodes and 80% probability of edges.
- **octree-D5**: 8-ary complete tree of depth 5 (root + five full levels of nodes).
- **int-16K**: 16K random integers in the range [0, 100000).
Conclusions

- Dynamic Parallelism
- Function Call Re-Vectorization
- Shuffle Nightmare

Diagram:

- Vertical axis: Programmability
- Horizontal axis: Efficiency
- Arrows indicating relationships between concepts:
  - Dynamic Parallelism to Function Call Re-Vectorization
  - Shuffle Nightmare to Function Call Re-Vectorization
QUESTIONS?

Email us:
Rubens Emilio Alves Moreira [rubens@dcc.ufmg.br]
Sylvain Collange [sylvain.collange@inria.fr]
Fernando Magno Quintão Pereira [fernando@dcc.ufmg.br]

Check our website:
http://cuda.dcc.ufmg.br/~swan