



# Function Call Re-Vectorization

**Pupil:** Rubens Emilio Alves Moreira

**Advisor:** Fernando Magno Quintão Pereira





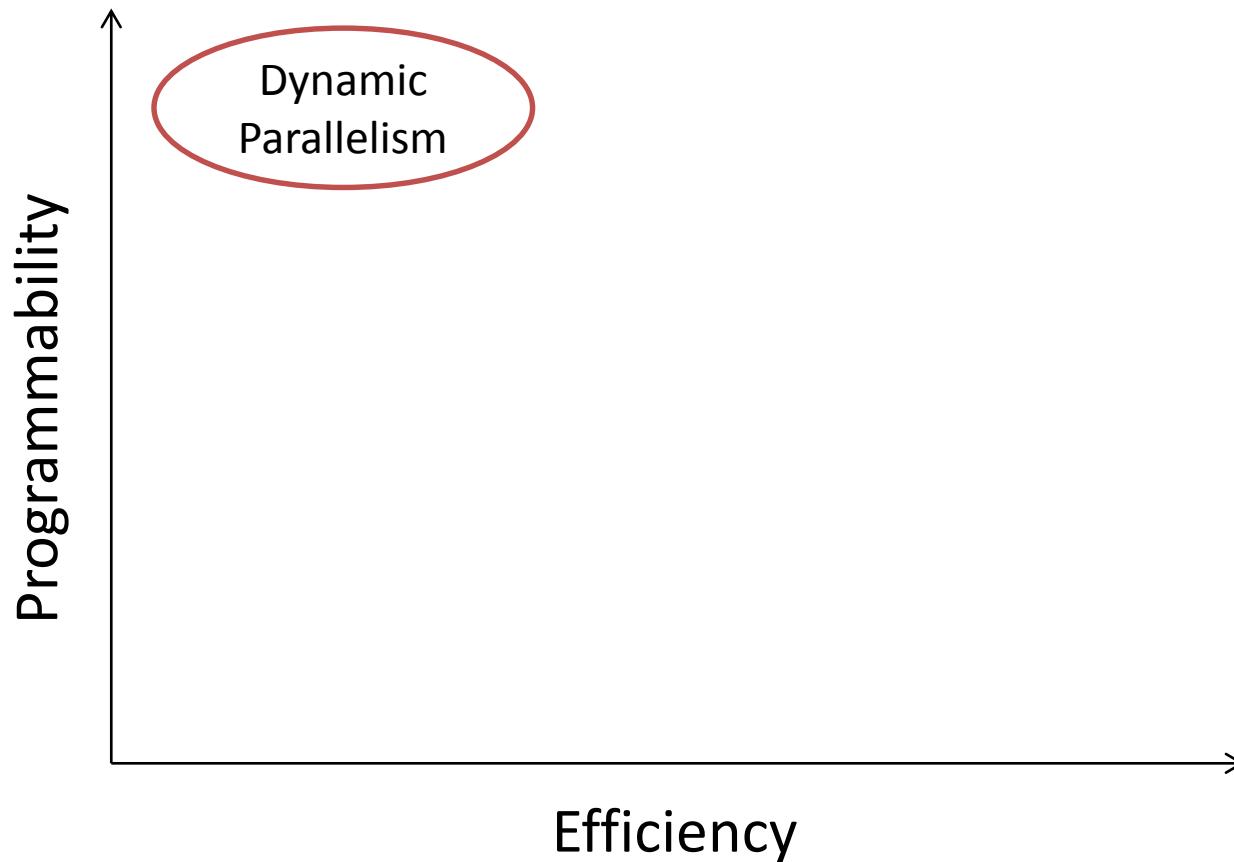
# Function Call Re-Vectorization





# Function Call Re-Vectorization

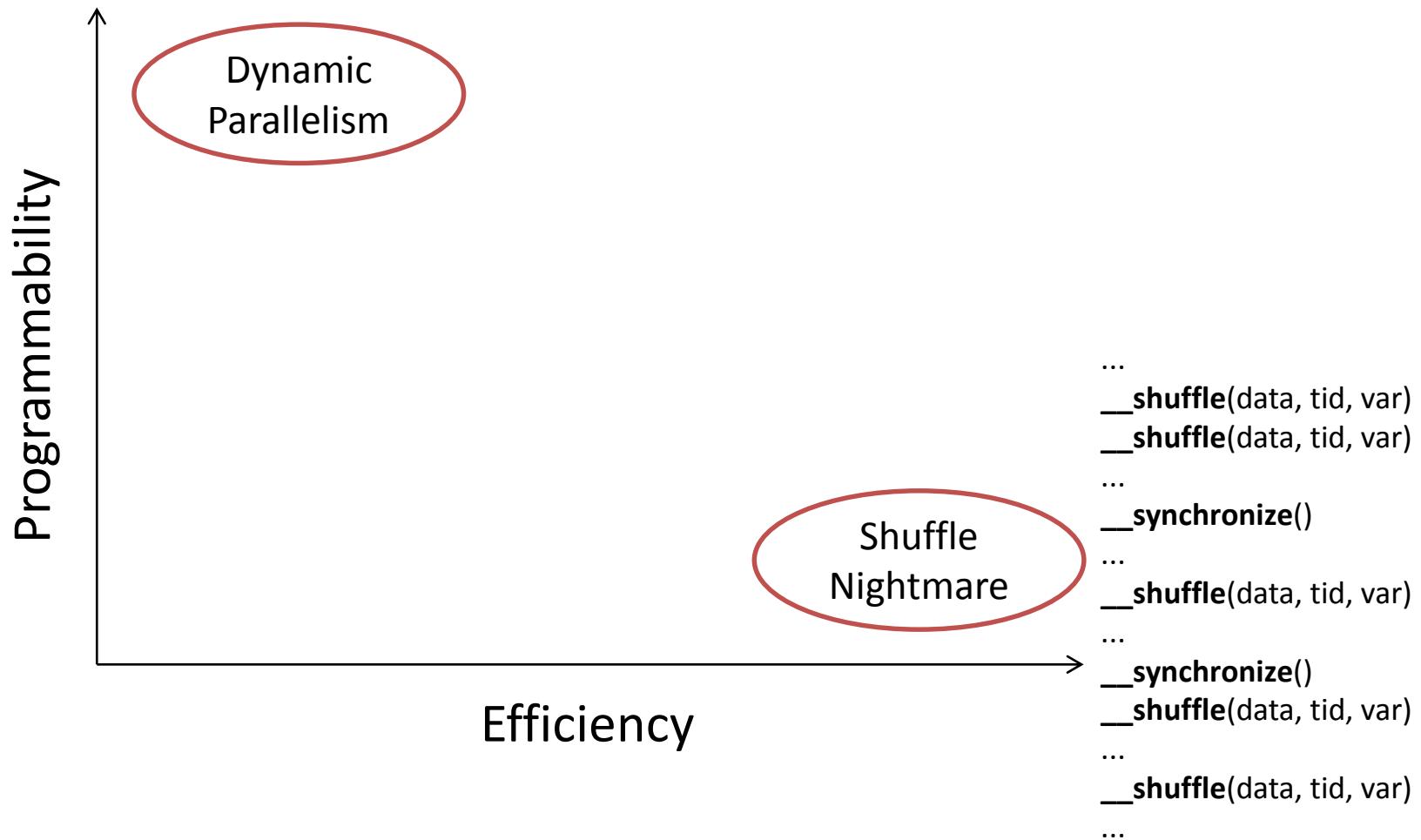
CUDA: kernel<<<#warps, #threads>>>(args...)





# Function Call Re-Vectorization

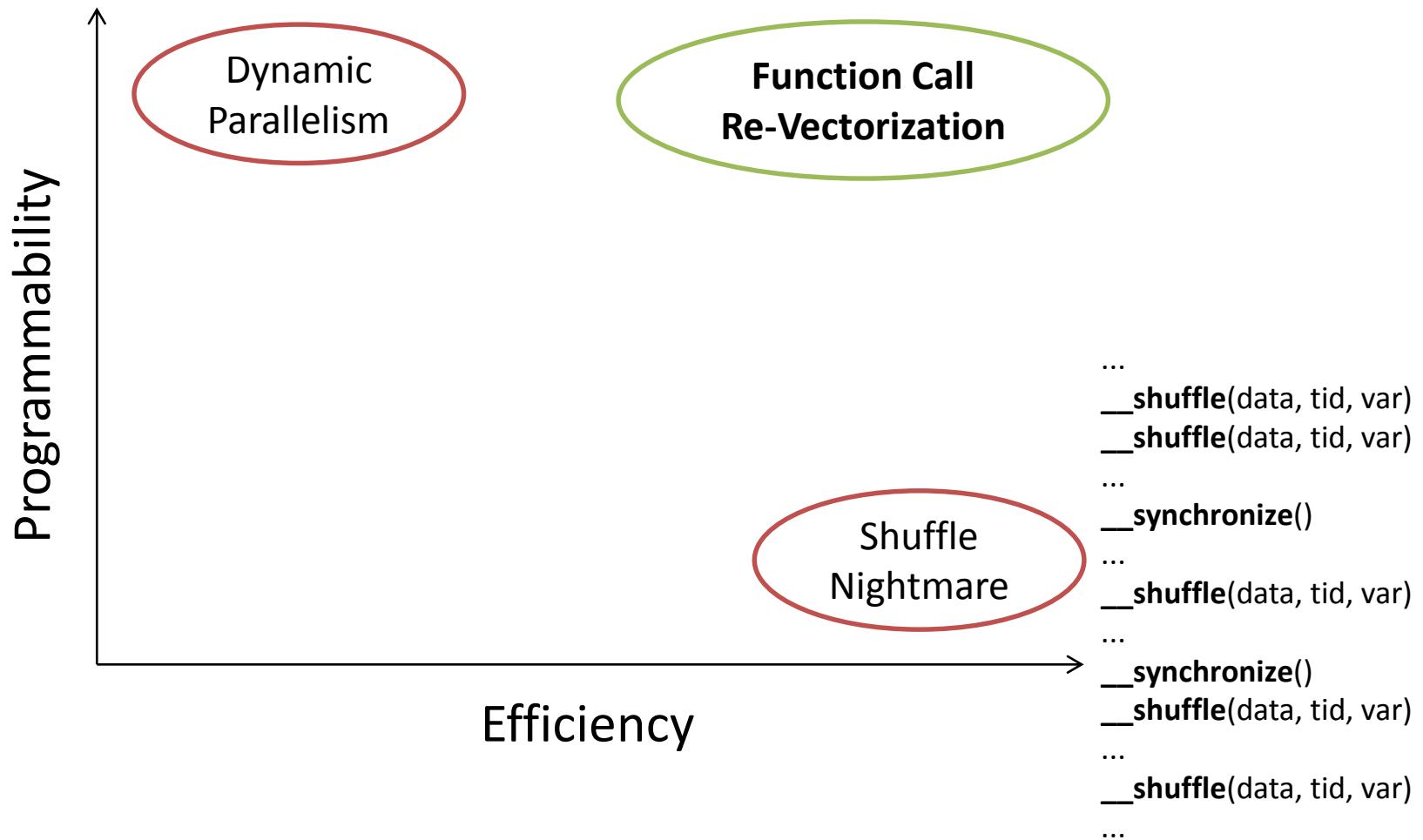
CUDA: kernel<<<#warps, #threads>>>(args...)





# Function Call Re-Vectorization

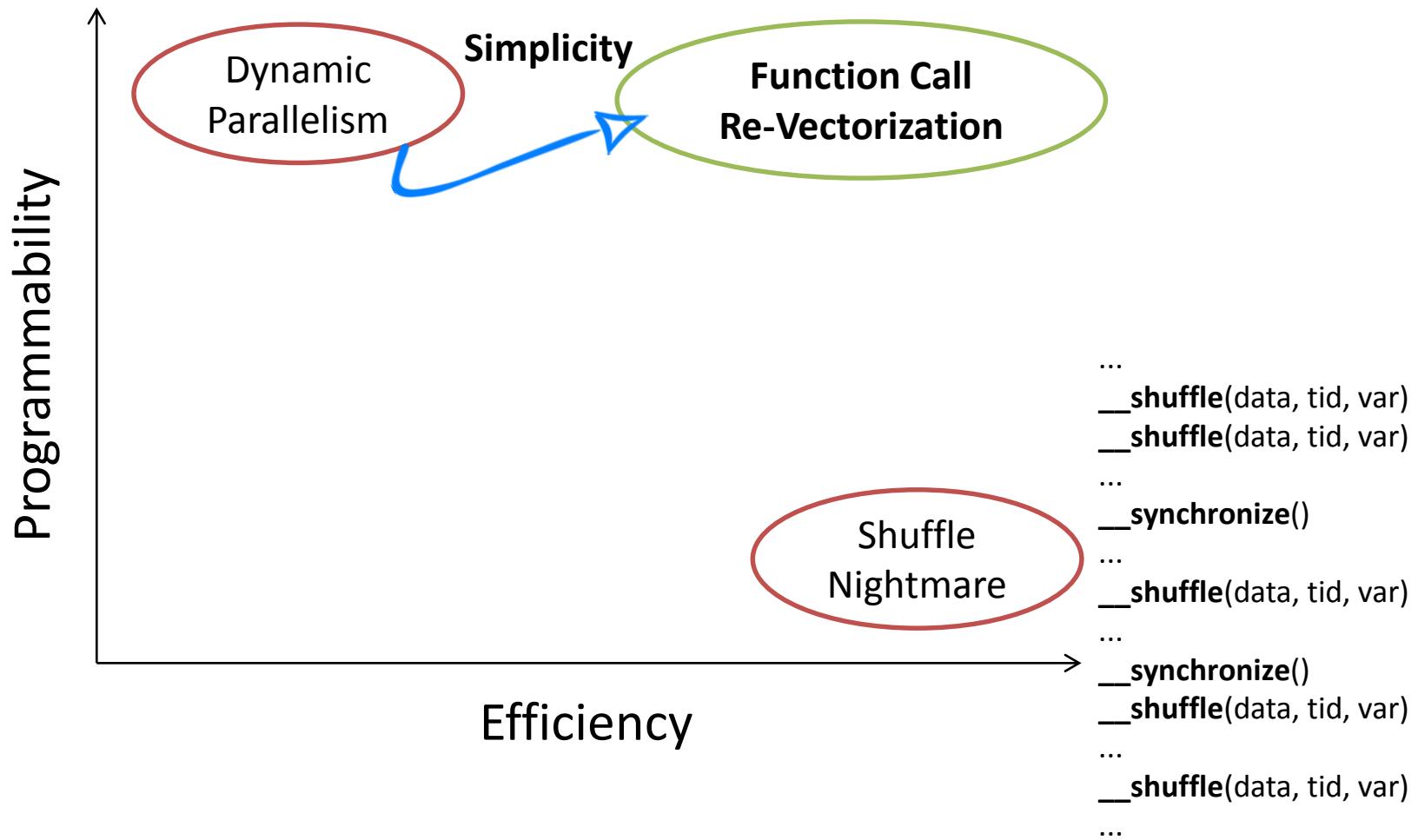
CUDA: kernel<<<#warps, #threads>>>(args...)





# Function Call Re-Vectorization

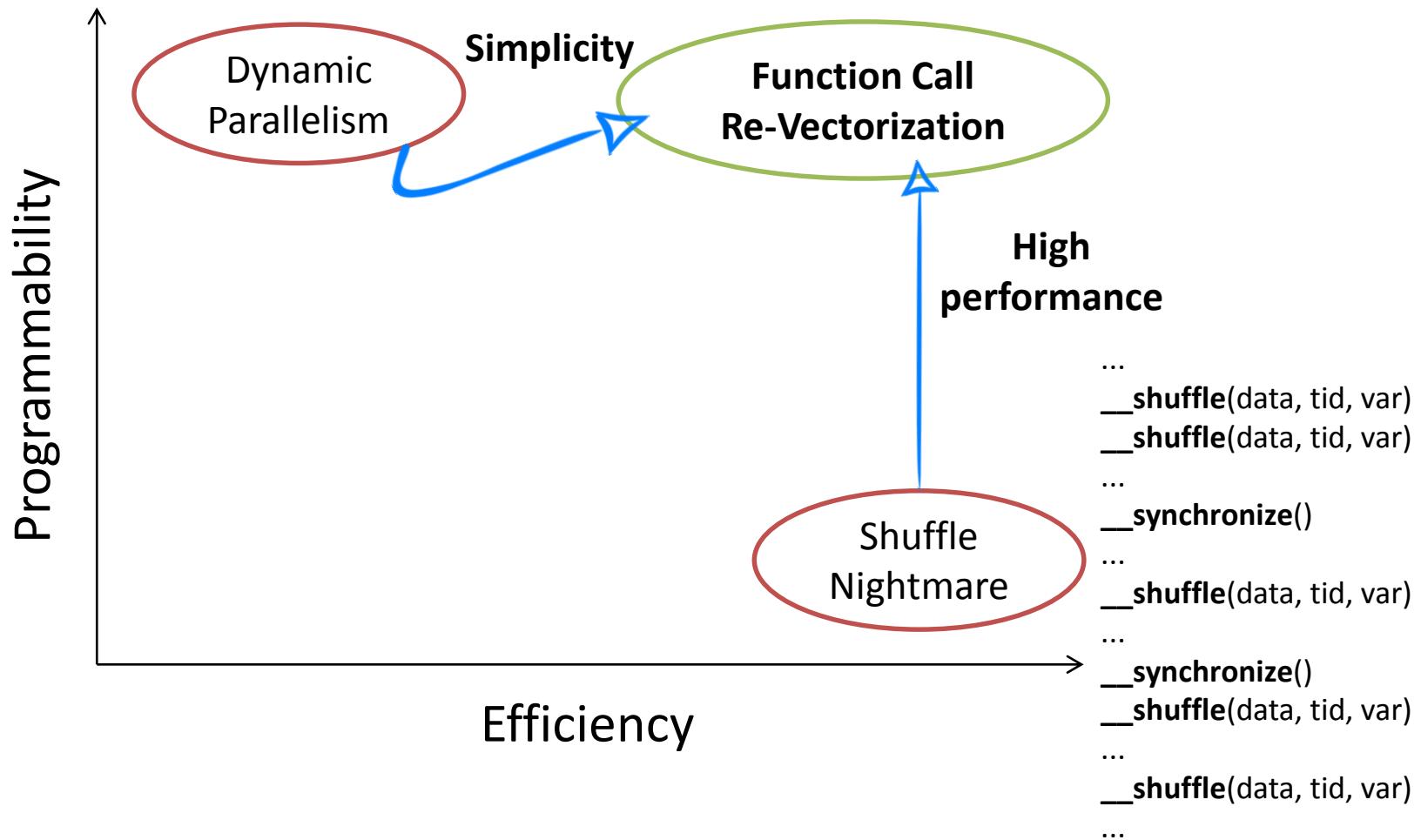
CUDA: kernel<<<#warps, #threads>>>(args...)





# Function Call Re-Vectorization

CUDA: kernel<<<#warps, #threads>>>(args...)





## Function Call Re-Vectorization

```
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.



## Function Call Re-Vectorization

```
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) {  
    memcpy<<<1, 4>>>(dest[tid], src[tid], N[tid]);  
}
```

CUDA's *nested kernel call: Dynamic parallelism*

Too much overhead



# Function Call Re-Vectorization

```
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) {  
    memcpy<<<1, 4>>>(dest[tid], src[tid], N[tid]);  
}
```

CUDA's *nested kernel call: Dynamic parallelism*

```
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);  
        }  
    }  
}
```

Warp-synchronous wrapper for SIMD memory copy.

Too much overhead

Too many lines of code



# Function Call Re-Vectorization

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

Simplicity + Performance, *a.k.a.*  
**CREV**

SIMD implementation of memory copy.

```
void memcpy_wrapper(int **dest, int **src, int *N, int mask) {  
    memcpy<<<1, 4>>>(dest[tid], src[tid], N[tid]);  
}
```

CUDA's *nested kernel call: Dynamic parallelism*

```
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);  
        }  
    }  
}
```

Warp-synchronous wrapper for SIMD memory copy.

Too much  
overhead

Too many  
lines of code



# Function Call Re-Vectorization

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 function calls  
within divergent regions*



DEPARTMENT OF COMPUTER SCIENCE  
UNIVERSIDADE FEDERAL DE MINAS GERAIS  
FEDERAL UNIVERSITY OF MINAS GERAIS, BRAZIL

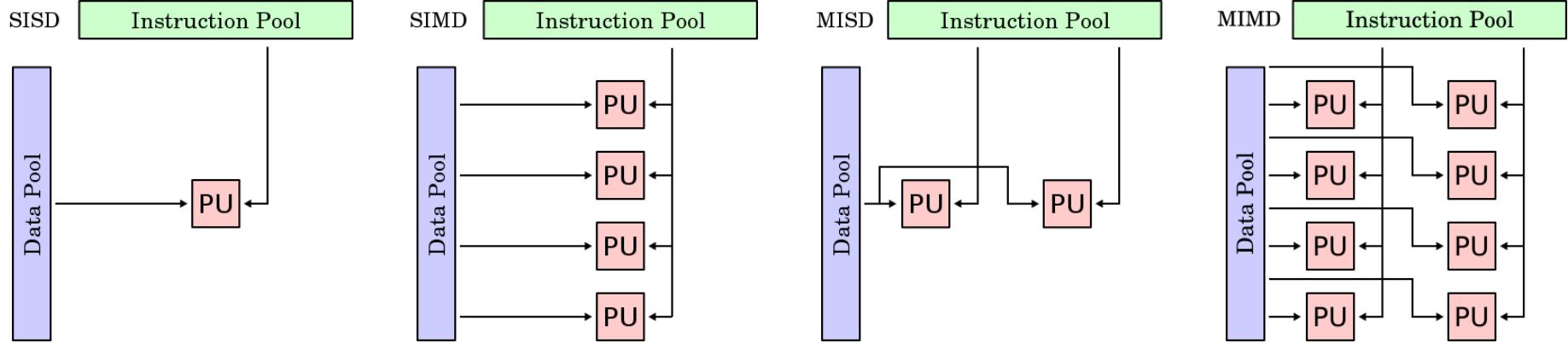
# CONCEPTS

---



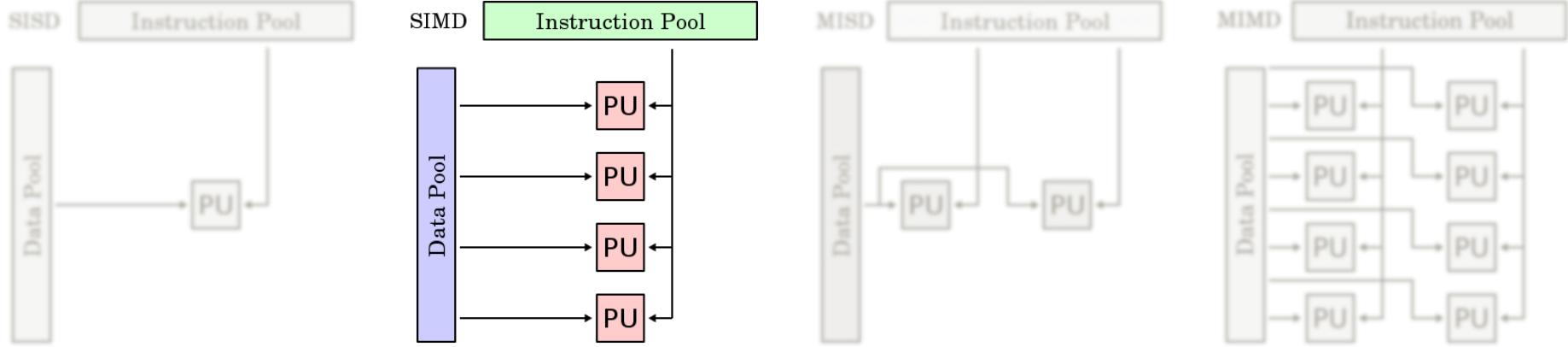


# Concepts: Flynn's Taxonomy





# Concepts: Flynn's Taxonomy

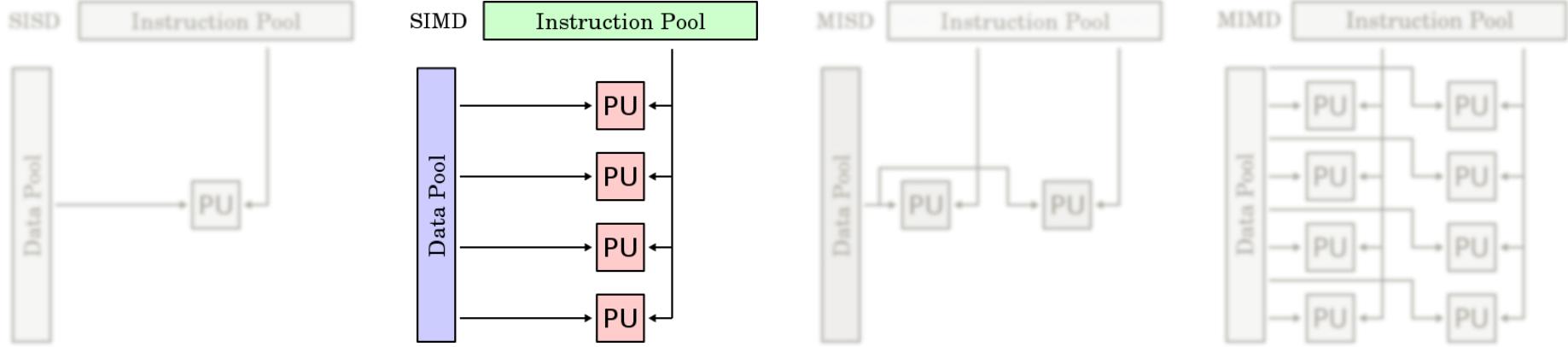


## SIMD (Single Instruction Multiple Data):

- One instruction fetcher
- Multiple processing units
  - Global memory bench
  - Private memory bench
- Lockstep execution



# Concepts: Flynn's Taxonomy



## SIMD (Single Instruction Multiple Data):

- One instruction fetcher
- Multiple processing units
  - Global memory bench
  - Private memory bench
- **Lockstep execution**

All processing units execute  
the **same instruction!**



## Concepts: Lockstep Execution

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

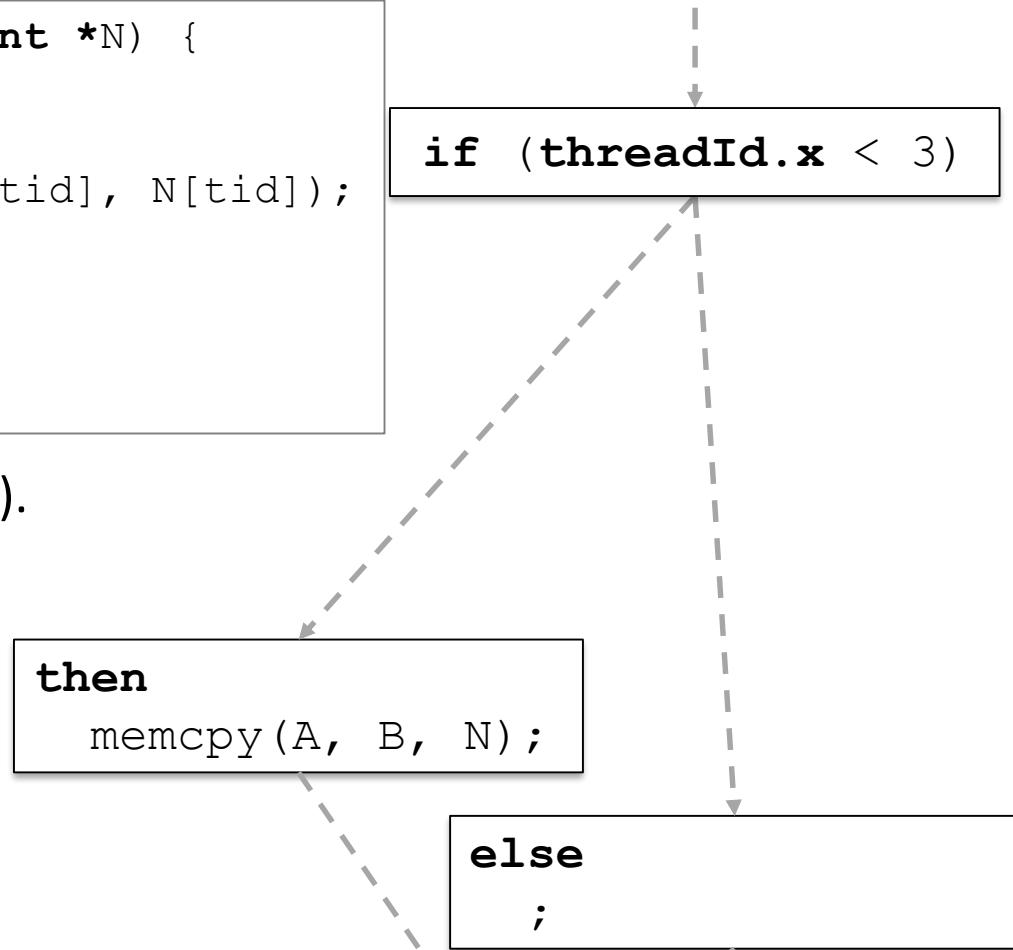
Kernel for parallel execution (CUDA).



## Concepts: Lockstep Execution

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

Kernel for parallel execution (CUDA).



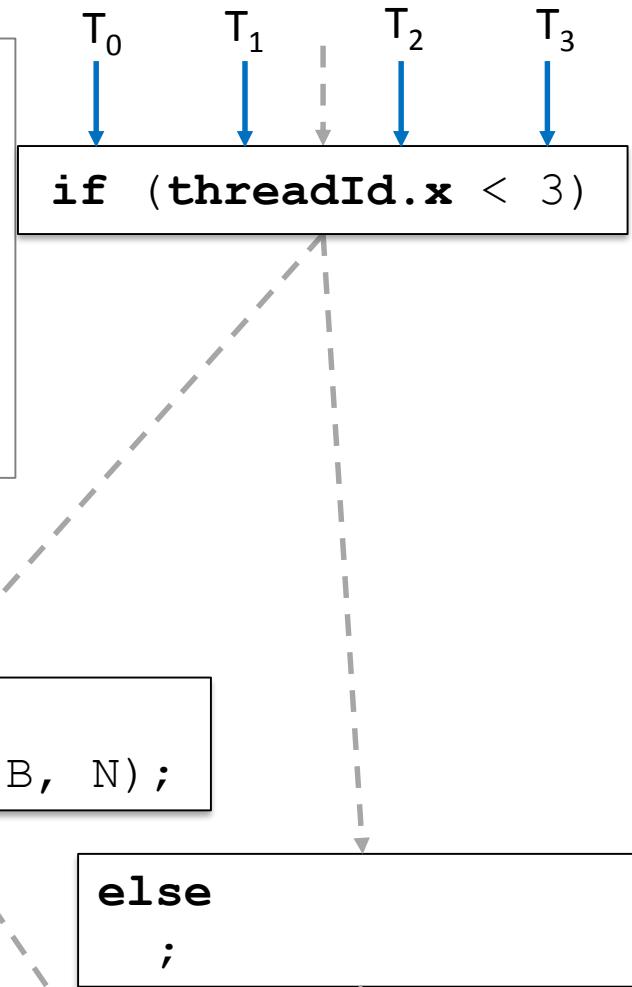
Control flow graph for kernel.



# Concepts: Lockstep Execution

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

Kernel for parallel execution (CUDA).



**SIMD: LOCKSTEP EXECUTION!**

Control flow graph for kernel.



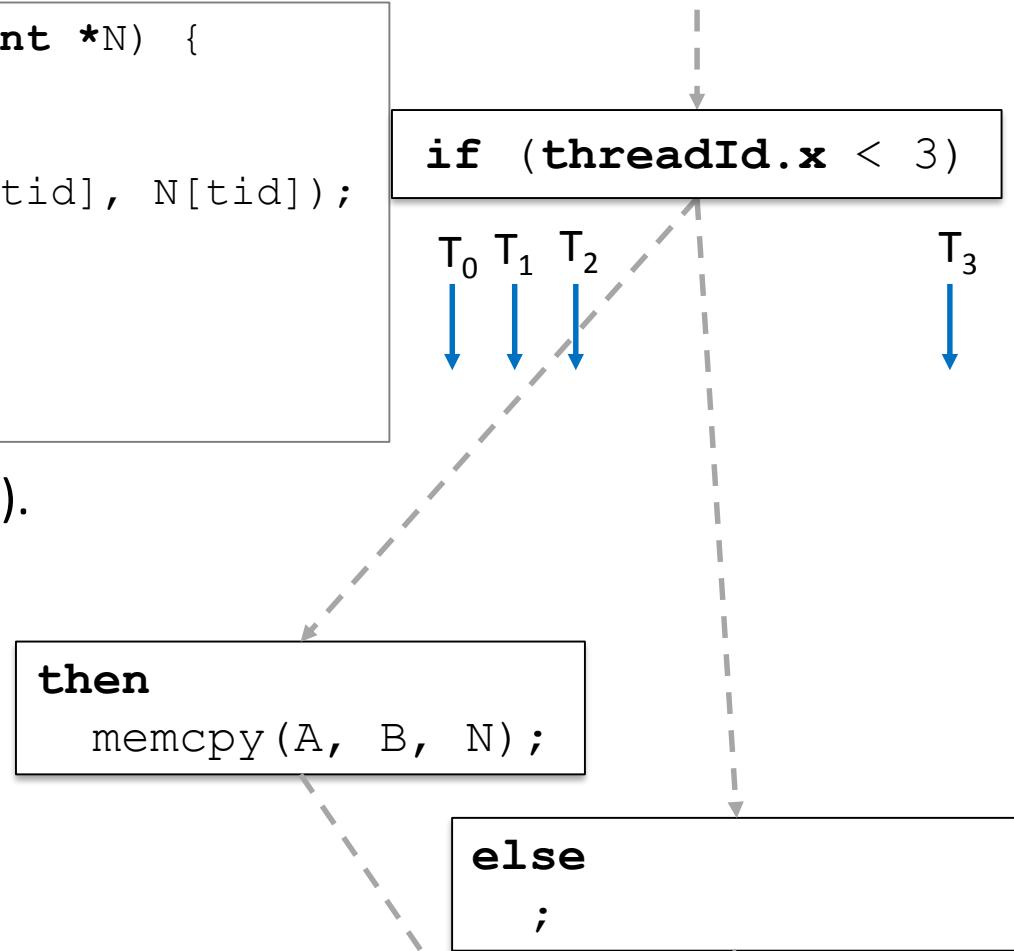
# Concepts: Lockstep Execution

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

Kernel for parallel execution (CUDA).

**SIMD: LOCKSTEP EXECUTION!**

Control flow graph for kernel.

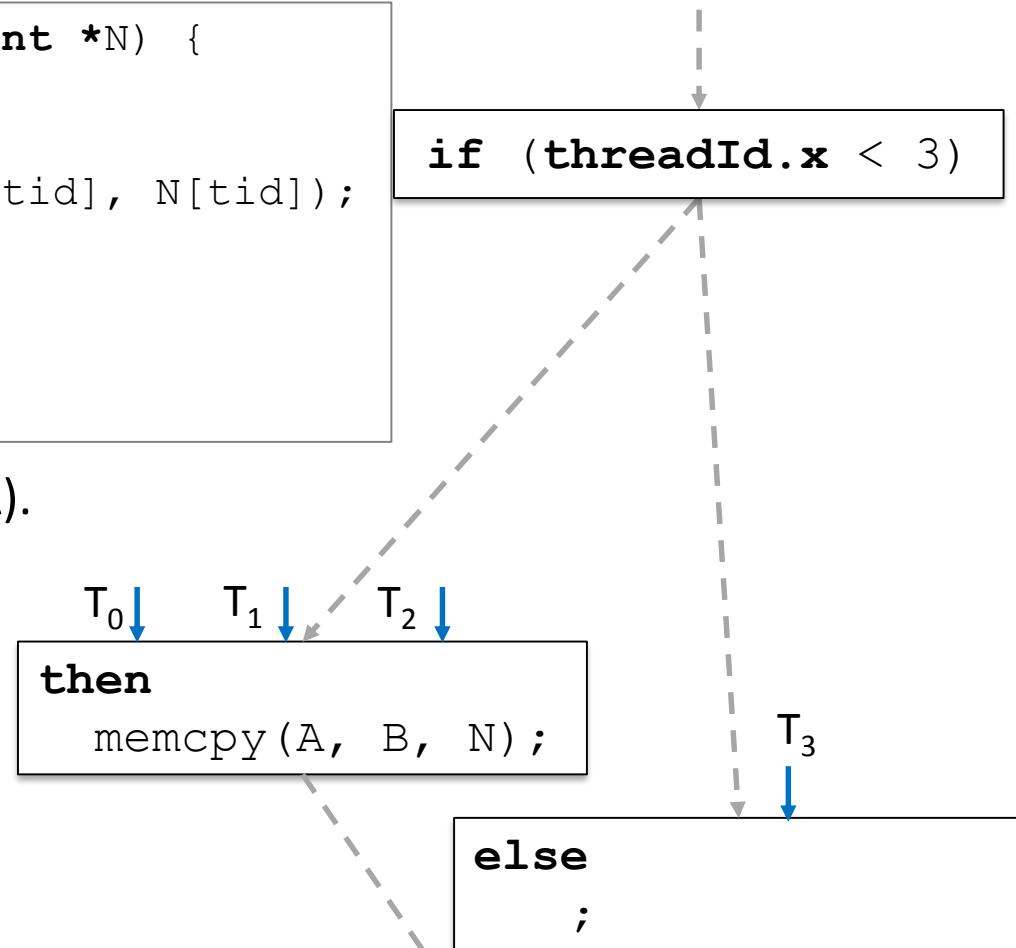




# Concepts: Lockstep Execution

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

Kernel for parallel execution (CUDA).



**SIMD: LOCKSTEP EXECUTION!**

Control flow graph for kernel.



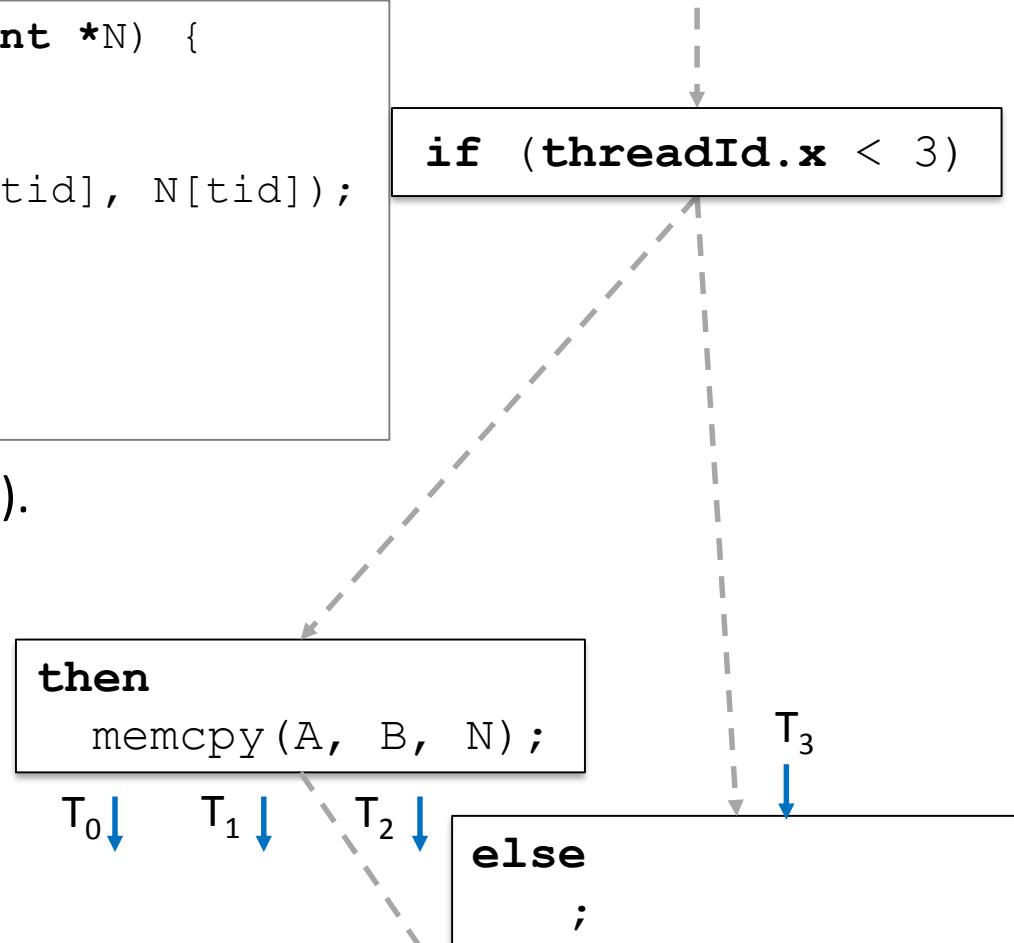
# Concepts: Lockstep Execution

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

Kernel for parallel execution (CUDA).

**SIMD: LOCKSTEP EXECUTION!**

Control flow graph for kernel.



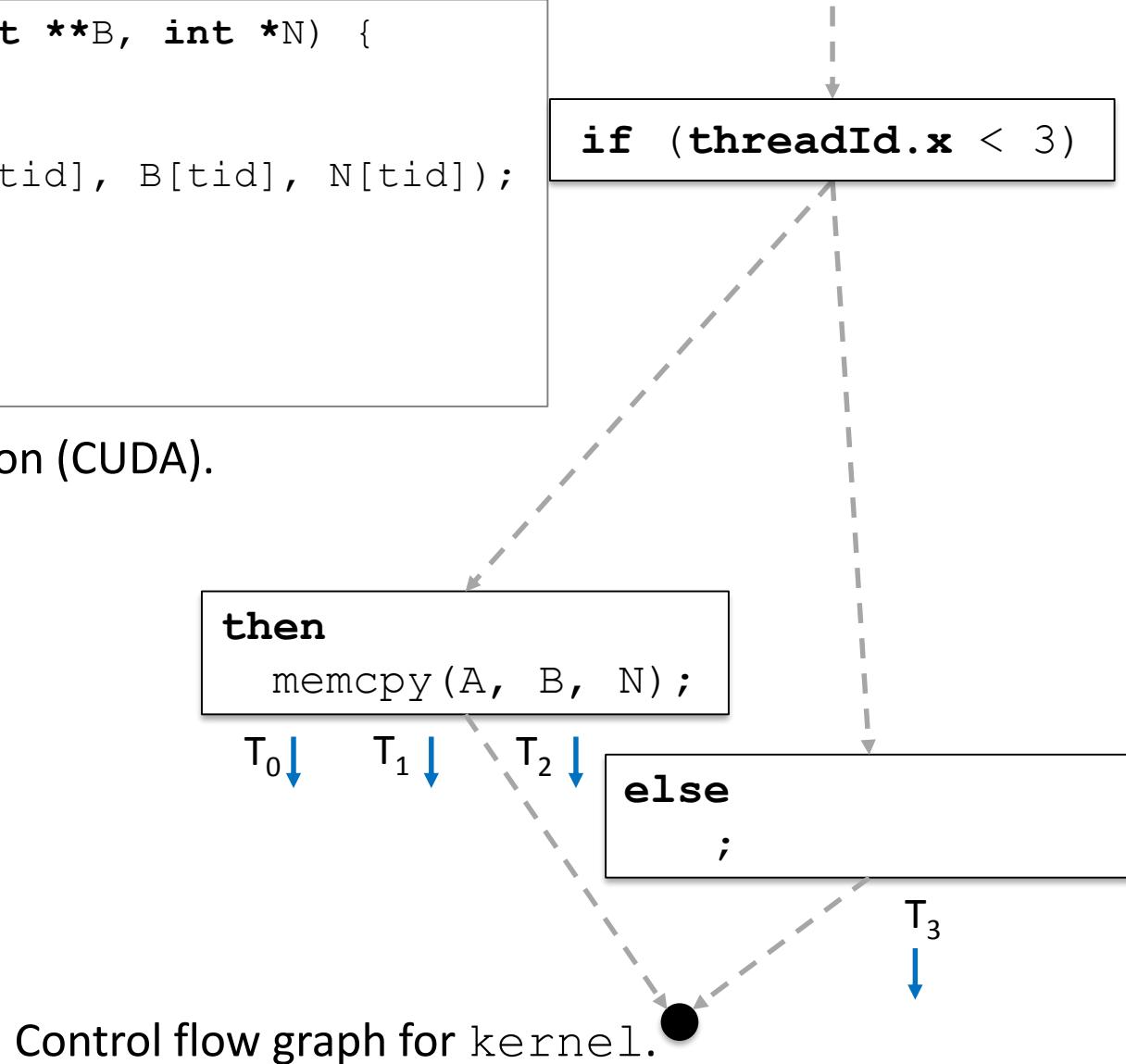


# Concepts: Lockstep Execution

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

Kernel for parallel execution (CUDA).

**SIMD: LOCKSTEP EXECUTION!**





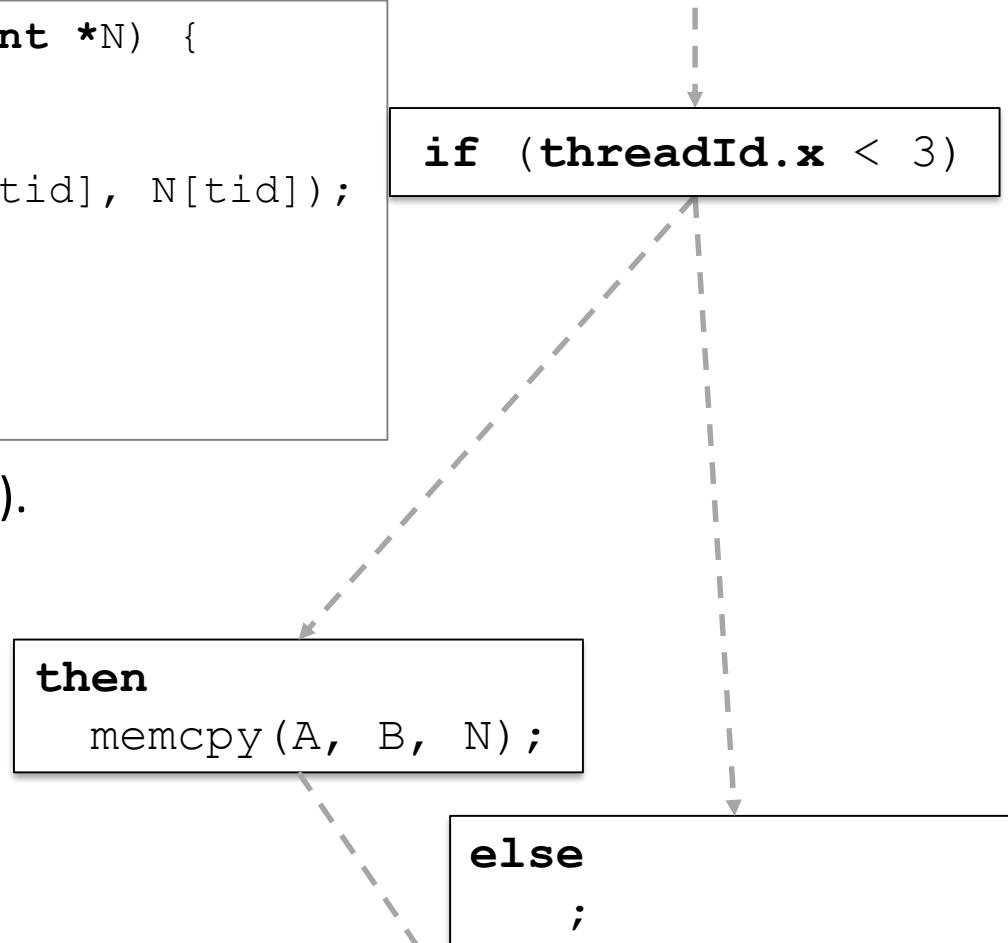
# Concepts: Lockstep Execution

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

Kernel for parallel execution (CUDA).

**SIMD: LOCKSTEP EXECUTION!**

Control flow graph for kernel.



$T_0 \downarrow T_1 \downarrow T_2 \downarrow T_3 \downarrow$



DEPARTMENT OF COMPUTER SCIENCE  
UNIVERSIDADE FEDERAL DE MINAS GERAIS  
FEDERAL UNIVERSITY OF MINAS GERAIS, BRAZIL

# DIVERGENCES

---

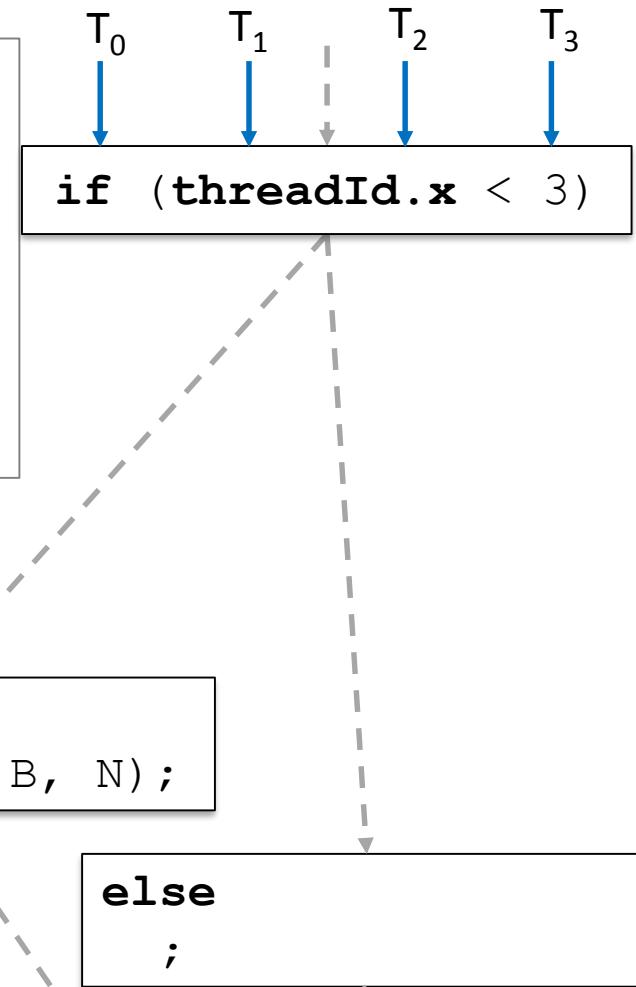




# Divergences

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

Kernel for parallel execution (CUDA).



**SIMD: LOCKSTEP EXECUTION!**

Control flow graph for kernel.

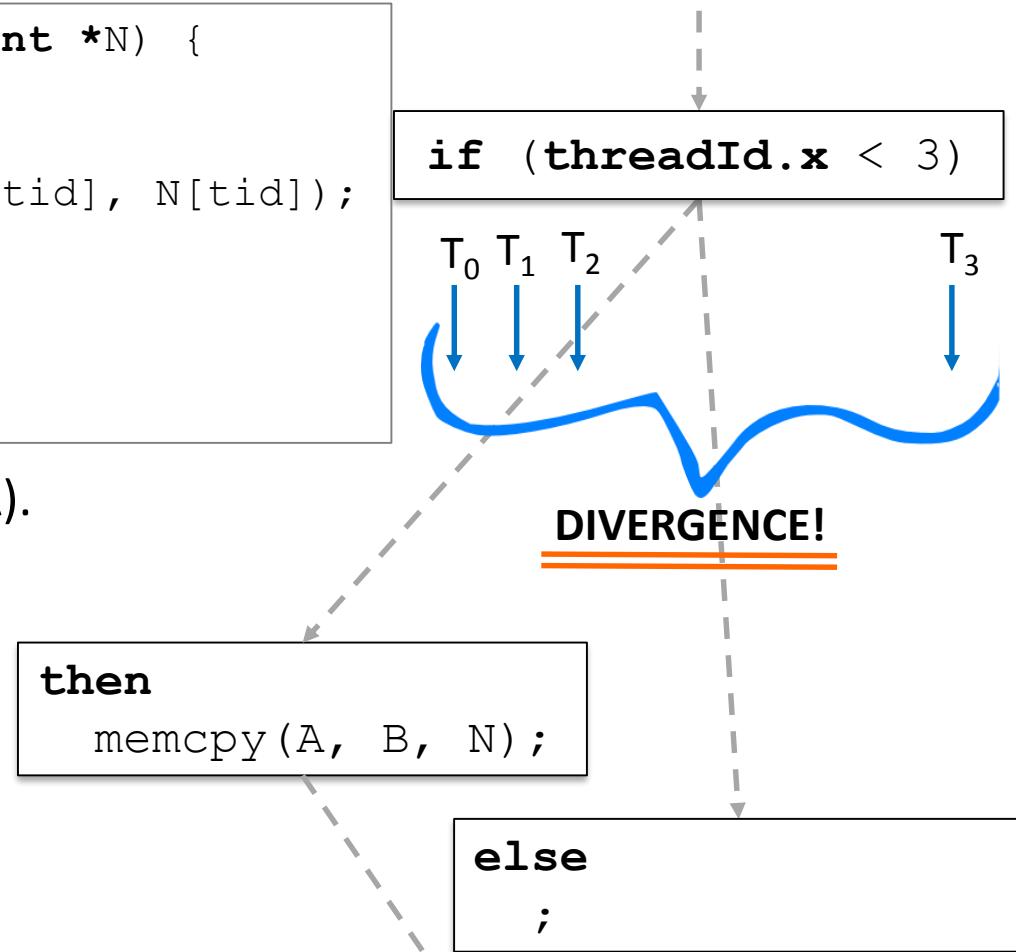


# Divergences

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

Kernel for parallel execution (CUDA).

**SIMD: LOCKSTEP EXECUTION!**



Control flow graph for kernel.

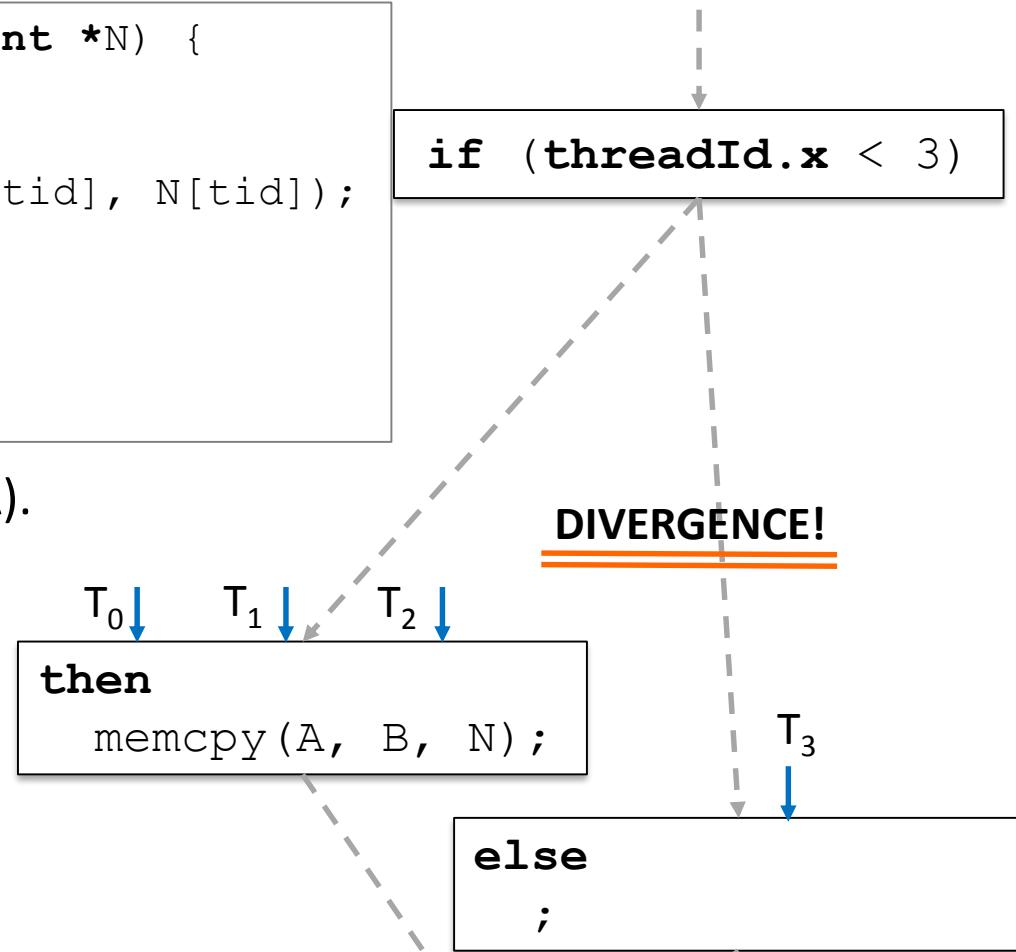


# Divergences

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

Kernel for parallel execution (CUDA).

**SIMD: LOCKSTEP EXECUTION!**



Control flow graph for kernel.

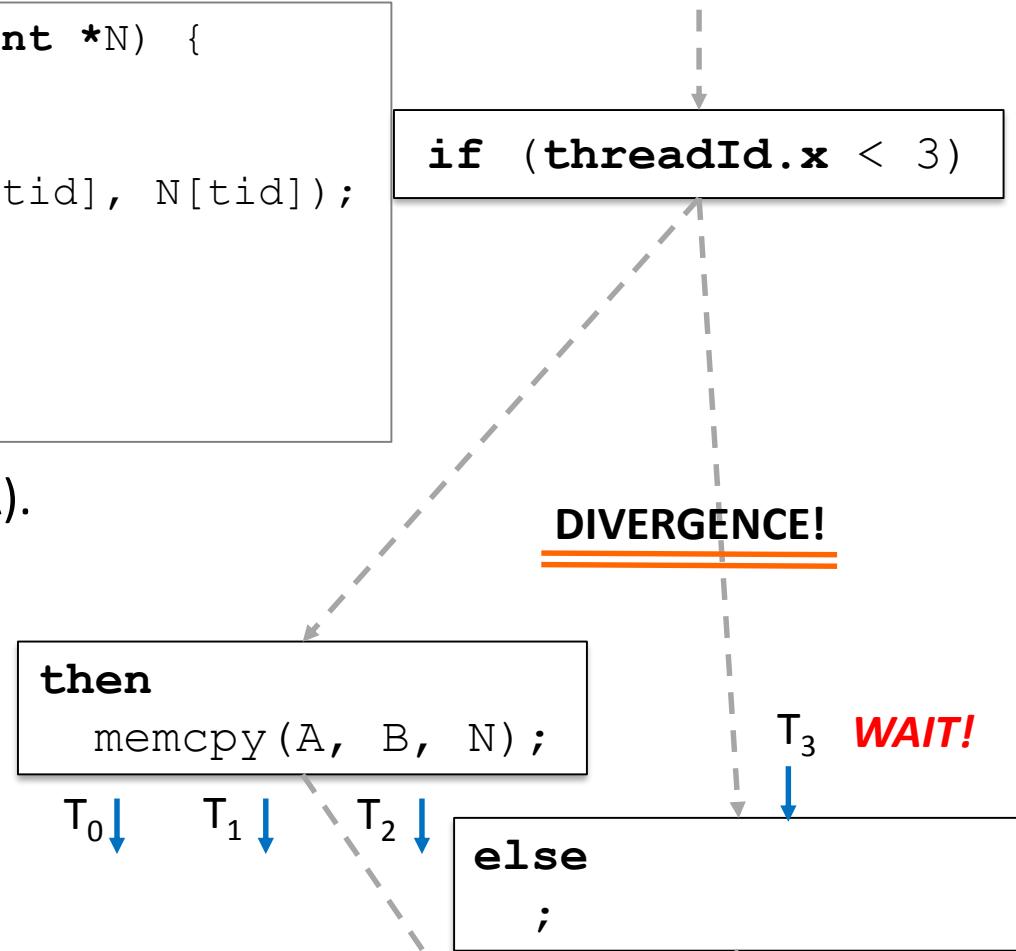


# Divergences

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

Kernel for parallel execution (CUDA).

**SIMD: LOCKSTEP EXECUTION!**



Control flow graph for kernel.

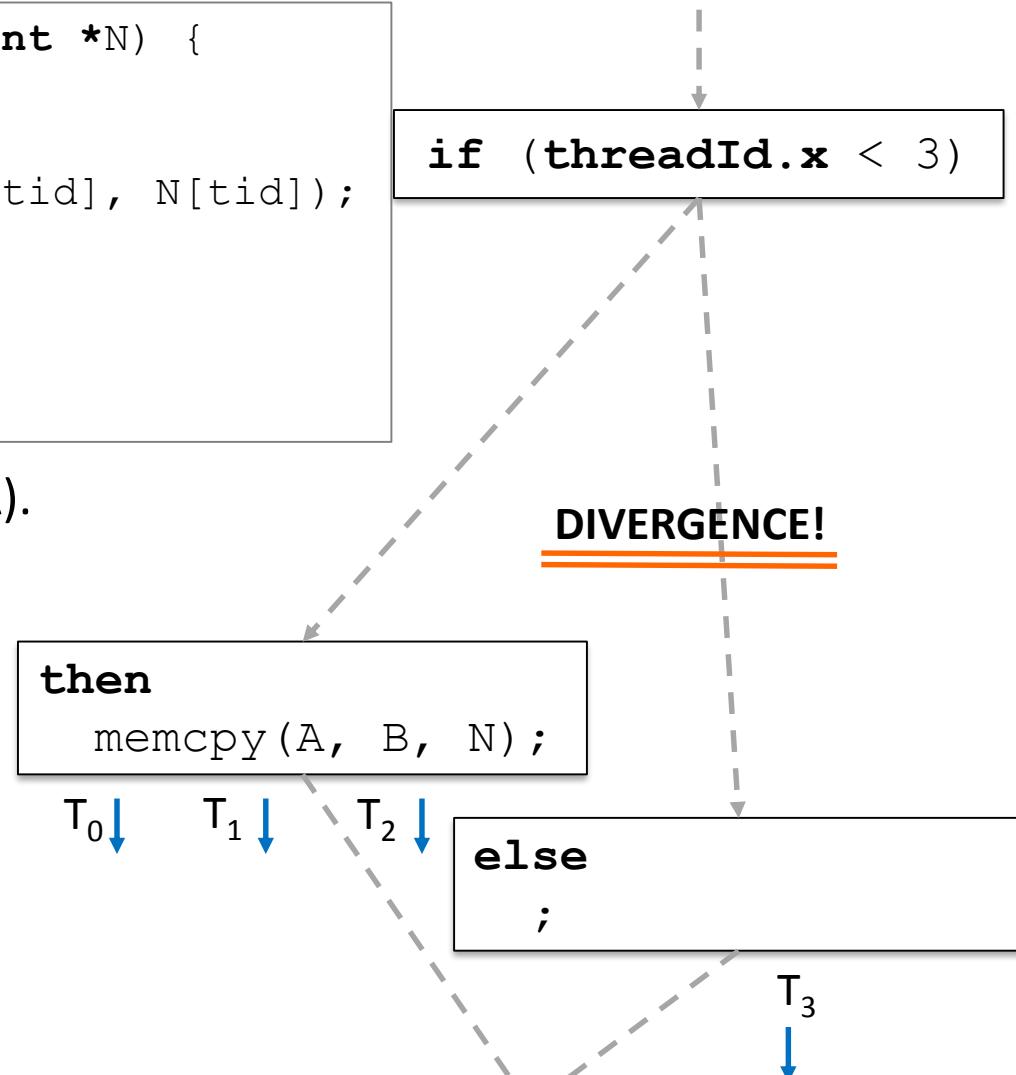


# Divergences

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

Kernel for parallel execution (CUDA).

**SIMD: LOCKSTEP EXECUTION!**



Control flow graph for kernel.



# Divergences

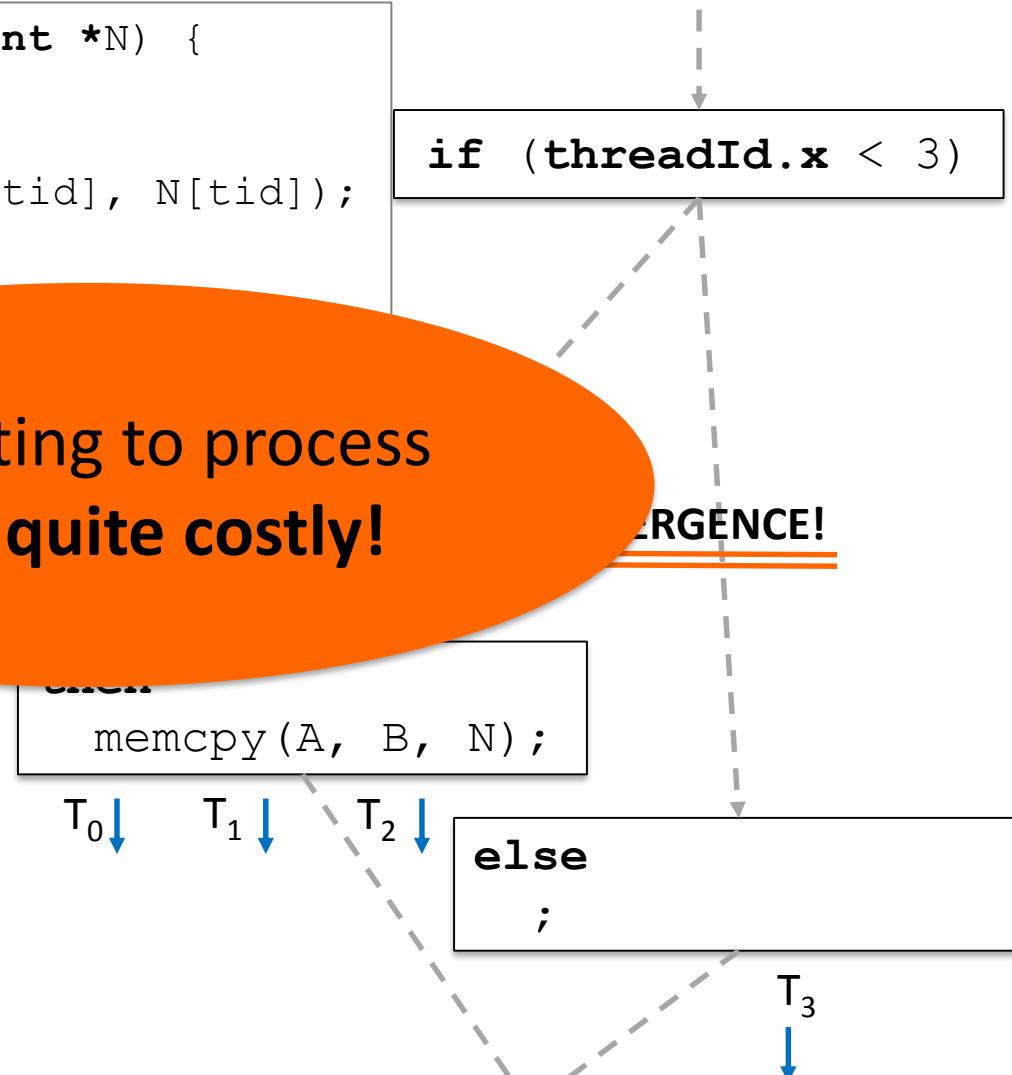
```
void kernel(int **A, int **B, int *N) {  
    int tid(threadId.x);  
    if (tid < 3) {  
        memcpy<<<1, 4>>>(A[tid], B[tid], N[tid]);  
    } else {  
        ;  
    }  
}
```

Kernel for parallel execution

And waiting to process  
can be **quite costly!**

**SIMD: LOCKSTEP EXECUTION!**

Control flow graph for kernel.





## Interlude: The Kernels of Samuel

```
__global__ void dec2zero(int *data, int N) {
    int xIndex = blockIdx.x * blockDim.x + threadIdx.x;
    if (xIndex < N) {
        while (data[xIndex] > 0) {
            data[xIndex]--;
        }
    }
}
```



## Interlude: The Kernels of Samuel

```
__global__ void dec2zero(int *data, int N) {
    int xIndex = blockIdx.x * blockDim.x + threadIdx.x;
    if (xIndex < N) {
        while (data[xIndex] > 0) {
            data[xIndex]--;
        }
    }
}
```



## Interlude: The Kernels of Samuel

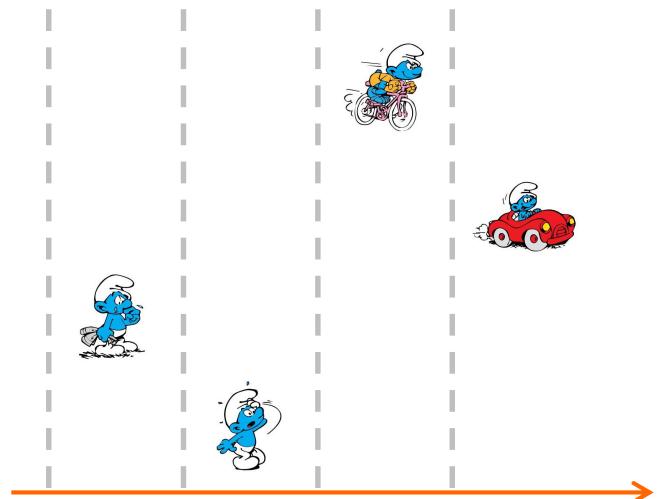
```
__global__ void dec2zero(int *data, int N) {
    int xIndex = blockIdx.x * blockDim.x + threadIdx.x;
    if (xIndex < N) {
        while (data[xIndex] > 0) {
            data[xIndex]--;
        }
    }
}
```



## Interlude: The Kernels of Samuel

Seeking for the lowest execution time,  
what is the best initialization of **data[]**?

```
__global__ void dec2zero(int *data, int N) {  
    int xIndex = blockIdx.x * blockDim.x + threadIdx.x;  
    if (xIndex < N) {  
        while (data[xIndex] > 0) {  
            data[xIndex]--;  
        }  
    }  
}
```

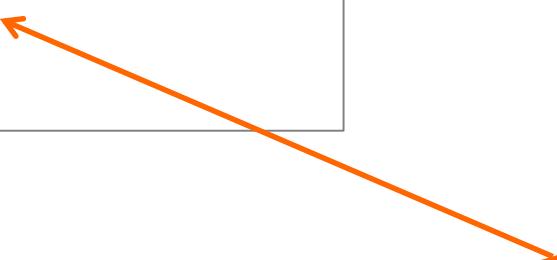




## Interlude: The Kernels of Samuel

```
int idx = threadIdx.x;
int dimx = blockDim.x;

void F(int *data, int size) {
    for (int i = idx; i < size; i += dimx) {
        data[i] = size - i + 1;
    }
}
```



F assigns the result of  
(size - i + 1) to data[i]



## Interlude: The Kernels of Samuel

```
int idx = threadIdx.x;
int dimx = blockDim.x;

void F(int *data, int size) {
    for (int i = idx; i < size; i += dimx) {
        data[i] = size - i + 1;
    }
}

void M(int *data, int size) {
    for (int i = idx; i < size; i += dimx) {
        data[i] = size; ←
    }
}
```

M assigns the constant value **size** to **data[i]**



## Interlude: The Kernels of Samuel

```
int idx = threadIdx.x;
int dimx = blockDim.x;

void F(int *data, int size) {
    for (int i = idx; i < size; i += dimx) {
        data[i] = size - i + 1;
    }
}

void M(int *data, int size) {
    for (int i = idx; i < size; i += dimx) {
        data[i] = size;
    }
}

void Q(int *data, int size) {
    for (int i = idx; i < size; i += dimx) {
        if (i % 2) data[i] = size;
    }
}
```

Q does also assign **size** to **data[i]**, but only for threads with odd index **i**



## Interlude: The Kernels of Samuel

```
int idx = threadIdx.x;
int dimx = blockDim.x;

void F(int *data, int size) {
    for (int i = idx; i < size; i += dimx) {
        data[i] = size - i + 1;
    }
}

void M(int *data, int size) {
    for (int i = idx; i < size; i += dimx) {
        data[i] = size;
    }
}

void Q(int *data, int size) {
    for (int i = idx; i < size; i += dimx) {
        if (i % 2) data[i] = size;
    }
}

void P(int *data, int size) {
    for (int i = idx; i < size; i += dimx) {
        data[i] = random() % size;
    }
}
```

P calls function **random** and assigns its value, modulo **size**, to **data[i]**





# Interlude: The Kernels of Samuel

```
int idx = threadIdx.x;
int dimx = blockDim.x;
```

```
void F(int *data, int size) {
    for (int i = idx; i < size; i += dimx) {
        data[i] = size - i + 1;
    }
}
```

```
void M(int *data, int size) {
    for (int i = idx; i < size; i += dimx) {
        data[i] = size;
    }
}
```

```
void Q(int *data, int size) {
    for (int i = idx; i < size; i += dimx) {
        if (i % 2) data[i] = size;
    }
}
```

```
void P(int *data, int size) {
    for (int i = idx; i < size; i += dimx) {
        data[i] = random() % size;
    }
}
```

```
__global__ void dec2zero(int *data, int N) {
    int xIndex = blockIdx.x * blockDim.x + threadIdx.x;
    if (xIndex < N) {
        while (data[xIndex] > 0) {
            data[xIndex]--;
        }
    }
}
```



## Interlude: The Kernels of Samuel

```
int idx = threadIdx.x;
int dimx = blockDim.x;
```

```
void F(int *data, int size) {
    for (int i = idx; i < size; i += dimx) {
        data[i] = size - i + 1;
    }
}
```

```
void M(int *data, int size) {
    for (int i = idx; i < size; i += dimx) {
        data[i] = size;
    }
}
```

```
void Q(int *data, int size) {
    for (int i = idx; i < size; i += dimx) {
        if (i % 2) data[i] = size;
    }
}
```

```
void P(int *data, int size) {
    for (int i = idx; i < size; i += dimx) {
        data[i] = random() % size;
    }
}
```

```
__global__ void dec2zero(int *data, int N) {
    int xIndex = blockIdx.x * blockDim.x + threadIdx.x;
    if (xIndex < N) {
        while (data[xIndex] > 0) {
            data[xIndex]--;
        }
    }
}
```

16153 $\mu$ s:  
all values are equal





## Interlude: The Kernels of Samuel

```
int idx = threadIdx.x;
int dimx = blockDim.x;
```

```
void F(int *data, int size) {
    for (int i = idx; i < size; i += dimx) {
        data[i] = size - i + 1;
    }
}
```

```
void M(int *data, int size) {
    for (int i = idx; i < size; i += dimx) {
        data[i] = size;
    }
}
```

```
void Q(int *data, int size) {
    for (int i = idx; i < size; i += dimx) {
        if (i % 2) data[i] = size;
    }
}
```

```
void P(int *data, int size) {
    for (int i = idx; i < size; i += dimx) {
        data[i] = random() % size;
    }
}
```

```
__global__ void dec2zero(int *data, int N) {
    int xIndex = blockIdx.x * blockDim.x + threadIdx.x;
    if (xIndex < N) {
        while (data[xIndex] > 0) {
            data[xIndex]--;
        }
    }
}
```

16250µs:  
values differ  
by constant



16153µs:  
all values are equal





# Interlude: The Kernels of Samuel

```
int idx = threadIdx.x;
int dimx = blockDim.x;

void F(int *data, int size) {
    for (int i = idx; i < size; i += dimx) {
        data[i] = size - i + 1;
    }
}

void M(int *data, int size) {
    for (int i = idx; i < size; i += dimx) {
        data[i] = size;
    }
}

void Q(int *data, int size) {
    for (int i = idx; i < size; i += dimx) {
        if (i % 2) data[i] = size;
    }
}

void P(int *data, int size) {
    for (int i = idx; i < size; i += dimx) {
        data[i] = random() % size;
    }
}
```

```
__global__ void dec2zero(int *data, int N) {
    int xIndex = blockIdx.x * blockDim.x + threadIdx.x;
    if (xIndex < N) {
        while (data[xIndex] > 0) {
            data[xIndex]--;
        }
    }
}
```

16250µs:  
values differ  
by constant



16153µs:  
all values are equal



30210µs:  
normal distribution  
of values





## Interlude: The Kernels of Samuel

```
int idx = threadIdx.x;
int dimx = blockDim.x;

void F(int *data, int size) {
    for (int i = idx; i < size; i += dimx) {
        data[i] = size - i + 1;
    }
}

void M(int *data, int size) {
    for (int i = idx; i < size; i += dimx) {
        data[i] = size;
    }
}

void Q(int *data, int size) {
    for (int i = idx; i < size; i += dimx) {
        if (i % 2) data[i] = size;
    }
}

void P(int *data, int size) {
    for (int i = idx; i < size; i += dimx) {
        data[i] = random() % size;
    }
}
```

```
__global__ void dec2zero(int *data, int N) {
    int xIndex = blockIdx.x * blockDim.x + threadIdx.x;
    if (xIndex < N) {
        while (data[xIndex] > 0) {
            data[xIndex]--;
        }
    }
}
```

16250µs:  
values differ  
by constant



16153µs:  
all values are equal



32193µs:  
half the values differ!



30210µs:  
normal distribution  
of values





# Interlude: The Kernels of Samuel

```
int id  
int dimx  
  
}  
  
} }  
  
Divergence is  
harmful to  
performance!  
  
void M(int *data, int size) {  
    for (int i = idx; i < size; i += dimx) {  
        data[i] = size;  
    }  
}  
  
void Q(int *data, int size) {  
    for (int i = idx; i < size; i += dimx) {  
        if (i % 2) data[i] = size;  
    }  
}  
  
void P(int *data, int size) {  
    for (int i = idx; i < size; i += dimx) {  
        data[i] = random() % size;  
    }  
}
```

```
__global__ void dec2zero(int *data, int N) {  
    int xIndex = blockIdx.x * blockDim.x + threadIdx.x;  
    if (xIndex < N) {  
        while (data[xIndex] > 0) {  
            data[xIndex]--;  
        }  
    }  
}
```

16250µs:  
values differ  
by constant



16153µs:  
all values are equal



32193µs:  
half the values differ!



30210µs:  
normal distribution  
of values



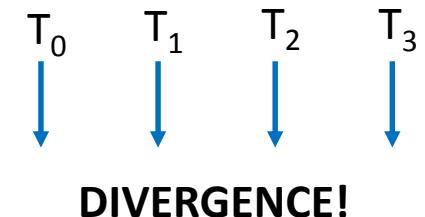


## Divergences: Coda

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

Kernel for parallel execution (CUDA).

Divergent region:  
only active threads  
run **memcpy**



FUNCTION **memcpy**

Control flow graph for memcpy.

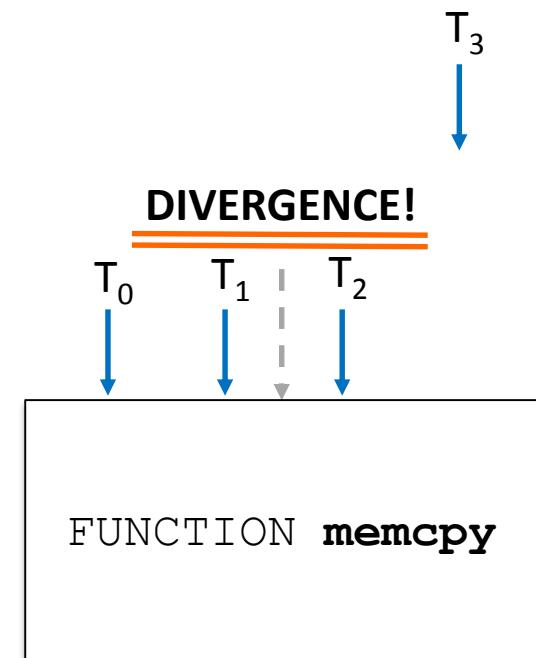


## Divergences: Coda

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

Kernel for parallel execution (CUDA).

Divergent region:  
only active threads  
run **memcpy**



Control flow graph for **memcpy**.



## Divergences: Coda

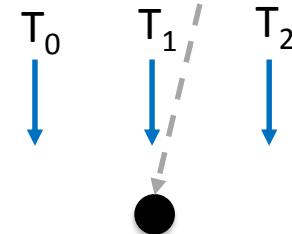
```
void kernel(int **A, int **B, int *N) {  
    int tid(threadId.x);  
    if (tid < 3) {  
        memcpy<<<1, 4>>>(A[tid], B[tid], N[tid]);  
    } else {  
        ;  
    }  
}
```

Kernel for parallel execution (CUDA).

Divergent region:  
only active threads  
run **memcpy**

**DIVERGENCE!**

FUNCTION **memcpy**



Control flow graph for **memcpy**.



# Divergences: Coda

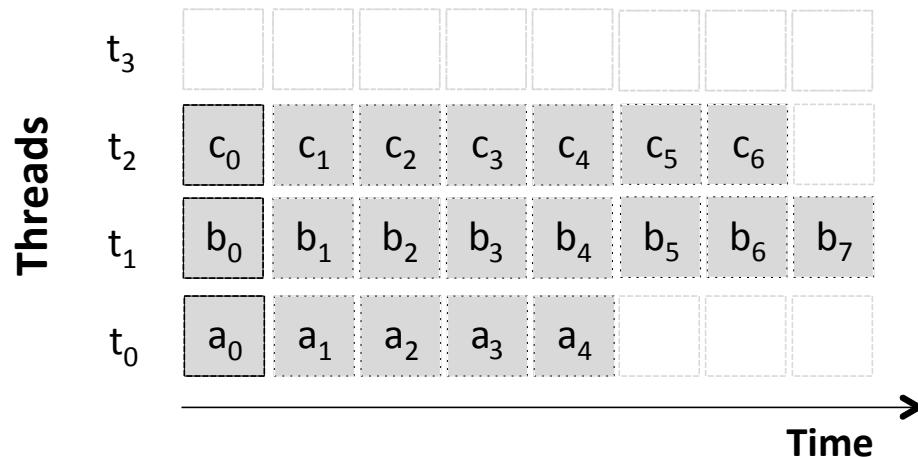
```
void kernel(int **A, int **B, int *N) {  
    int tid(threadId.x);  
    if (tid < 3) {  
        memcpy<<<1, 4>>>(A[tid], B[tid], N[tid]);  
    } else {  
        ;  
    }  
}
```

Kernel for parallel execution.

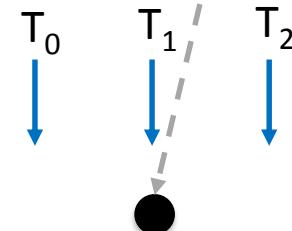
Suboptimal behavior:  
thread **T<sub>3</sub>** is inactive.  
Right?

**DIVERGENCE!**

Observed behavior:



FUNCTION **memcpy**



Control flow graph for memcpy.



# Divergences: Coda

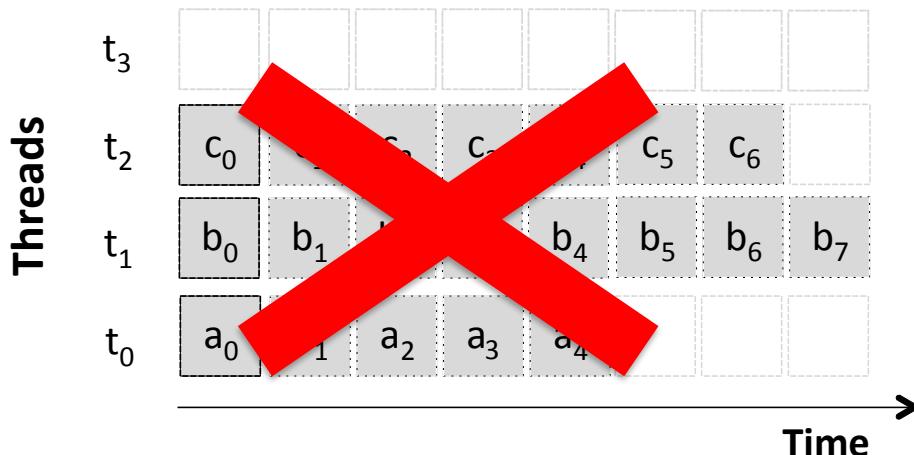
```
void kernel(int **A, int **B, int *N) {  
    int tid(threadId.x);  
    if (tid < 3) {  
        memcpy<<<1, 4>>>(A[tid], B[tid], N[tid]);  
    } else {  
        ;  
    }  
}
```

Kernel for parallel execution

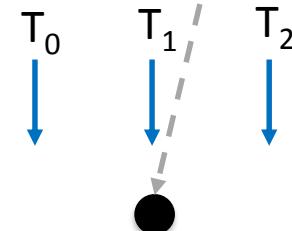
Not really! We are using  
**Dynamic Parallelism**

**DIVERGENCE!**

Observed behavior:



FUNCTION **memcpy**



Control flow graph for **memcpy**.



DEPARTMENT OF COMPUTER SCIENCE  
UNIVERSIDADE FEDERAL DE MINAS GERAIS  
FEDERAL UNIVERSITY OF MINAS GERAIS, BRAZIL

# DYNAMIC PARALLELISM

---





# Dynamic Parallelism

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

Kernel for parallel execution (CUDA)

CUDA's **nested kernel call**:  
**kernel<<<#warps, #threads>>>(args...)**



# Dynamic Parallelism

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

Kernel for parallel execution (CUDA)

CUDA's **nested kernel call**:  
`kernel<<<#warps, #threads>>>(args...)`

Launches a **new kernel**, with **all threads active**,  
to process the target function



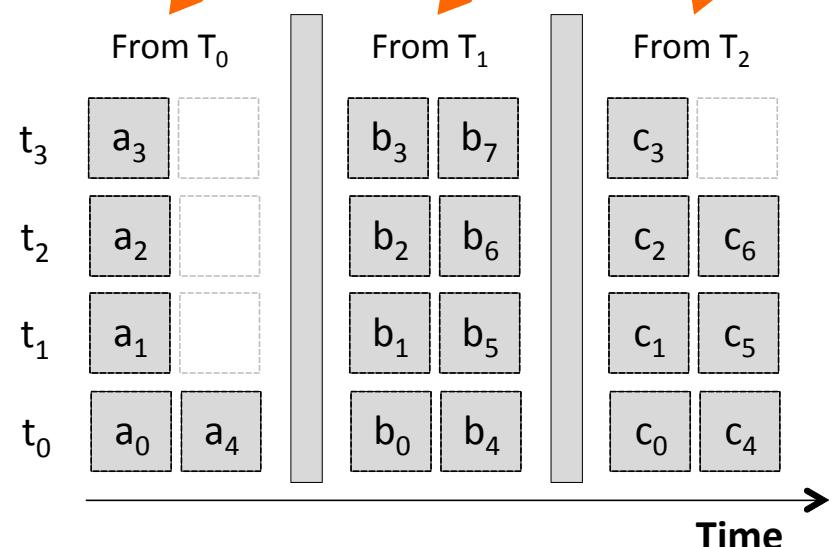
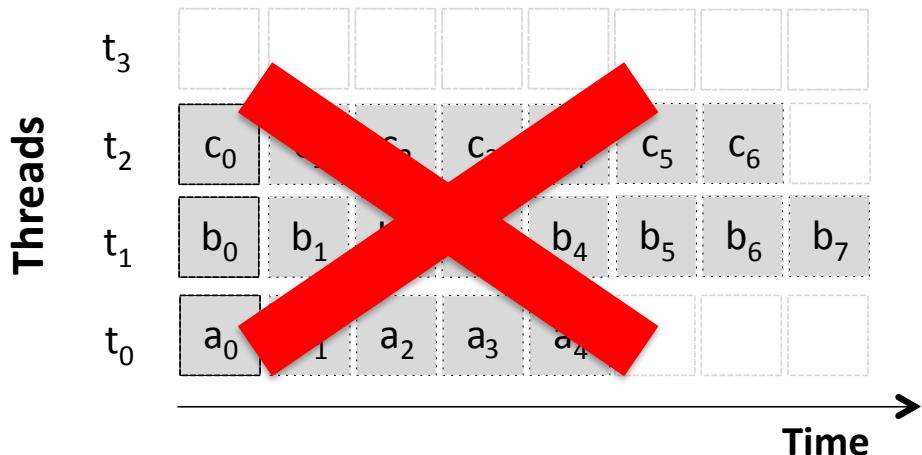
# Dynamic Parallelism

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

Kernel for parallel execution (CUDA).

memcpy runs once per active thread at memcpy<<<1, 4>>> call site!

Actual behavior with CUDA's dynamic parallelism:





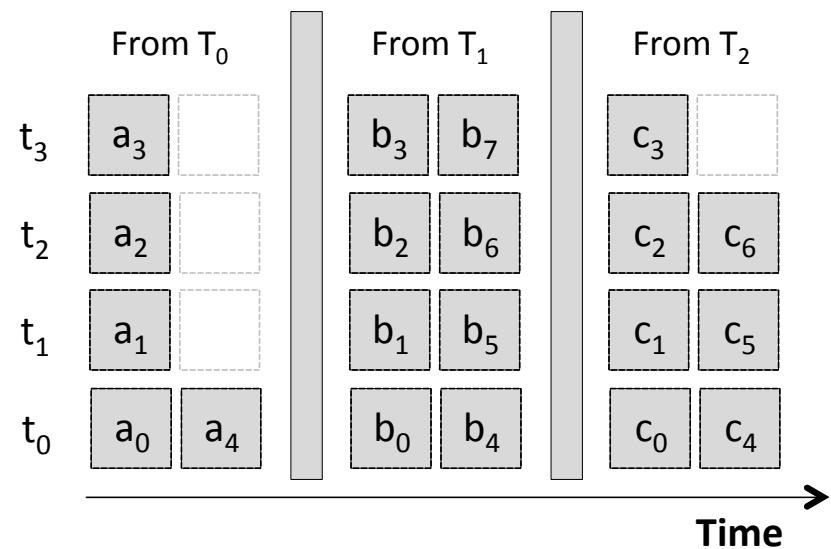
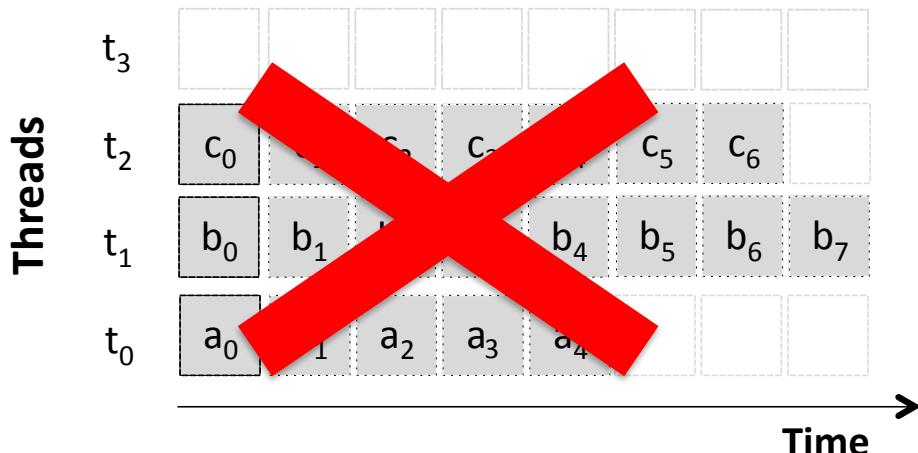
# Dynamic Parallelism

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

SIMD  
kernels!

Kernel for parallel execution (CUDA).

Actual behavior with CUDA's dynamic parallelism:



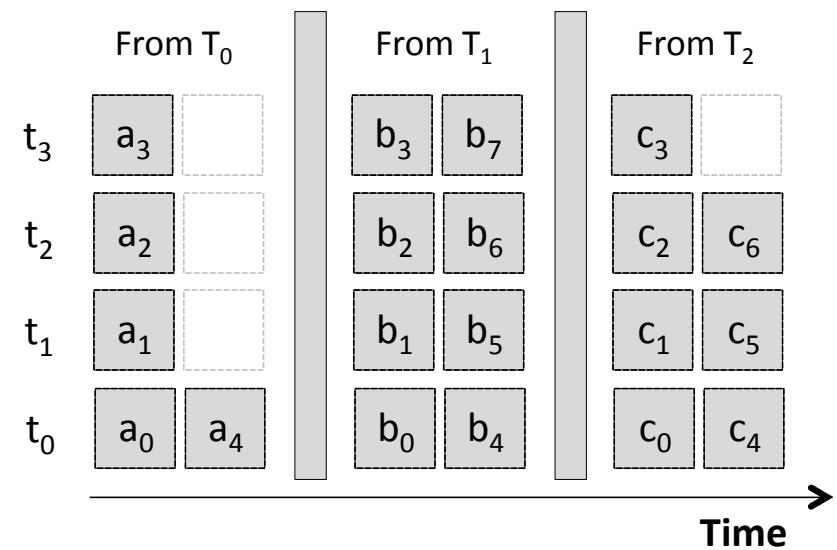
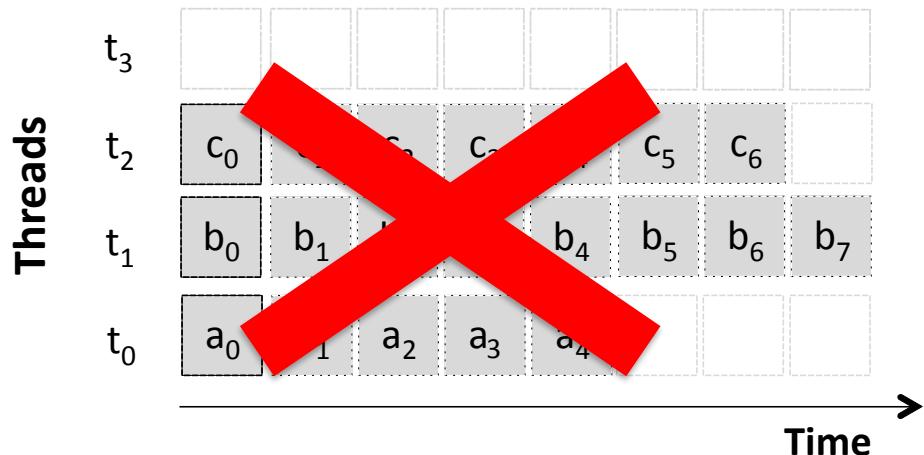


# Dynamic Parallelism

```
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.

Actual behavior with CUDA's dynamic parallelism:





# Dynamic Parallelism

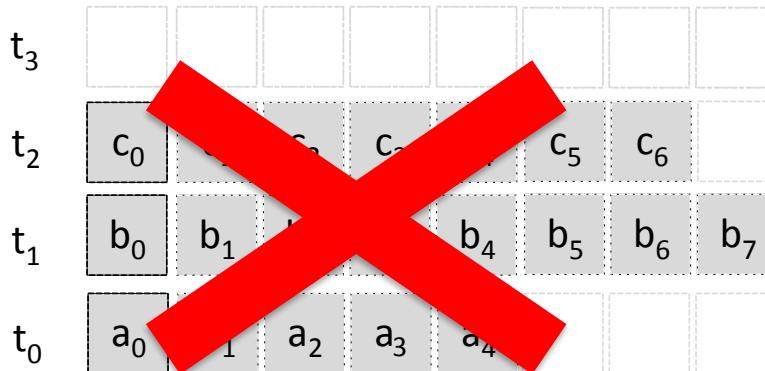
```
void memcpy(int *dest, int *src, int N) {  
    for (int i=threadId.x; i < N; i+=threadDim.x) {  
        dest[i] = src[i];  
    }  
}
```

SIMD implementation of memory copy.

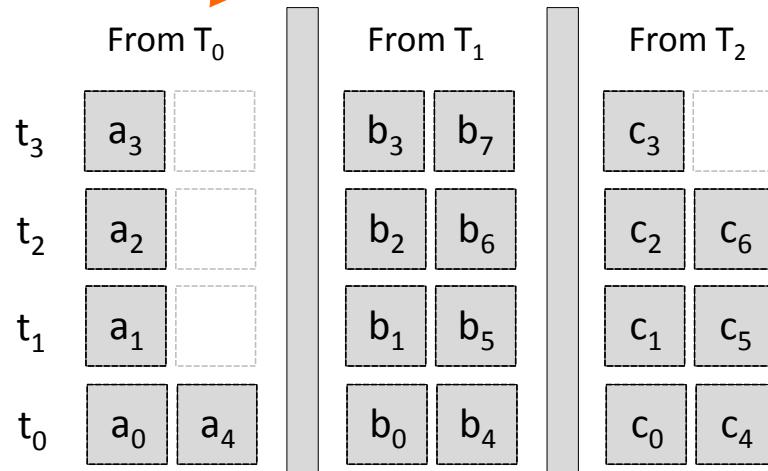
All threads work on a single vector!

Actual behavior with CUDA's dynamic parallelism:

Threads



Time



Time



# Dynamic Parallelism

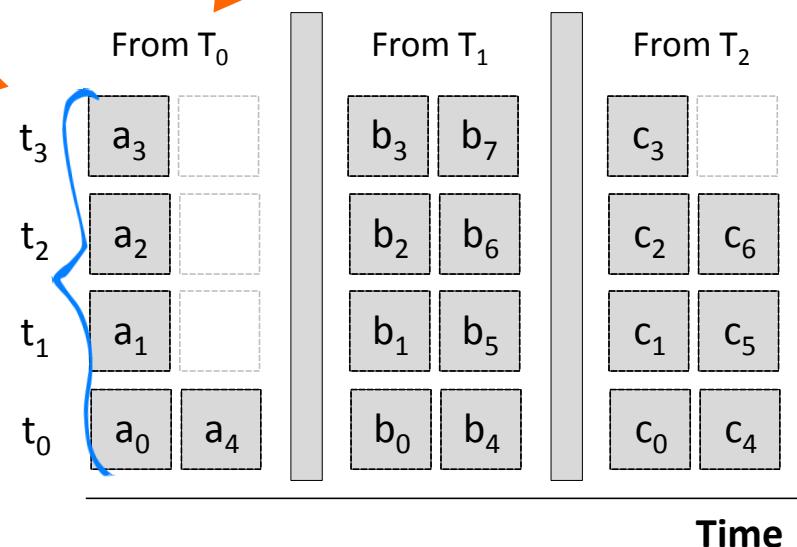
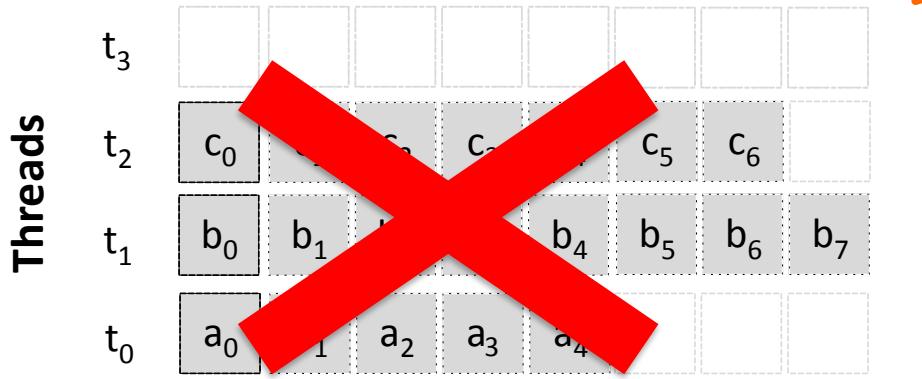
```
void memcpy(int *dest, int *src, int N) {  
    for (int i=threadId.x; i < N; i+=threadDim.x) {  
        dest[i] = src[i];  
    }  
}
```

SIMD implementation of memory copy.

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

All threads work on a single vector!

Actual behavior with CUDA's dynamic parallelism:





# Dynamic Parallelism

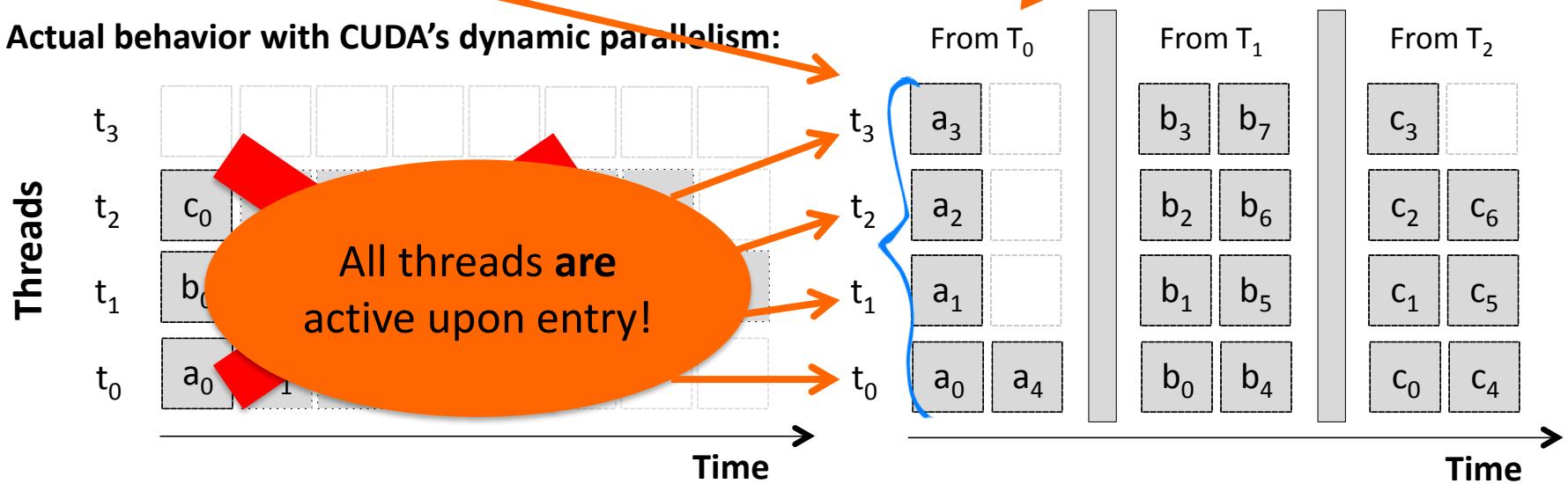
```
void memcpy(int *dest, int *src, int N) {  
    for (int i=threadId.x; i < N; i+=threadDim.x) {  
        dest[i] = src[i];  
    }  
}
```

SIMD implementation of memory copy.

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

All threads work on a single vector!

Actual behavior with CUDA's dynamic parallelism:





## Dynamic Parallelism

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

CUDA's Dynamic  
Parallelism:  
**Nested kernel calls**

Kernel for parallel execution (CUDA).



# Dynamic Parallelism

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

Kernel for parallel execution (CUDA).

CUDA's Dynamic Parallelism:  
**Nested kernel calls**

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



# Dynamic Parallelism

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

Kernel for parallel execution (CUDA).

CUDA's Dynamic Parallelism:  
**Nested kernel calls**

kernel<<<**#warps, #threads**>>>(args...);

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



# Dynamic Parallelism

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

Kernel for parallel execution (CUDA).

CUDA's Dynamic Parallelism:  
Nested kernel calls

kernel<<<**#warps**, **#threads**>>>(args...);

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

Parallel Time  $\sim$  Kernel Launching Overhead +  $\frac{\text{Sequential Time}}{\#warps \times \#threads}$

# 8 Dynamic Parallelism

```
void k  
int i  
if  
me ei  
}  
}
```

Kerne

**Important benefits** when new work is invoked within an executing GPU program include **removing the burden on the programmer to marshal and transfer the data** on which to operate. **Additional parallelism can be exposed** to the GPU's hardware schedulers and load balancers dynamically, **adapting in response to data-driven decisions** or workloads. **Algorithms and programming patterns** that had previously required modifications to eliminate recursion, irregular loop structure, or other constructs that do not fit a flat, single-level of parallelism **can be more transparently expressed.**

## Dynamic Parallelism in CUDA

ke

Source: [http://developer.download.nvidia.com/assets/cuda/files/CUDADownloads/TechBrief\\_Dynamic\\_Parallelism\\_in\\_CUDA.pdf](http://developer.download.nvidia.com/assets/cuda/files/CUDADownloads/TechBrief_Dynamic_Parallelism_in_CUDA.pdf)

$$\text{Parallel Time} \sim \text{Kernel Launching Overhead} + \frac{\text{Sequential Time}}{\# \text{warps} \times \# \text{threads}}$$

Has the overhead of allocating and scheduling a new kernel



DEPARTMENT OF COMPUTER SCIENCE  
UNIVERSIDADE FEDERAL DE MINAS GERAIS  
FEDERAL UNIVERSITY OF MINAS GERAIS, BRAZIL

# WARP-SYNCHRONOUS PROGRAMMING

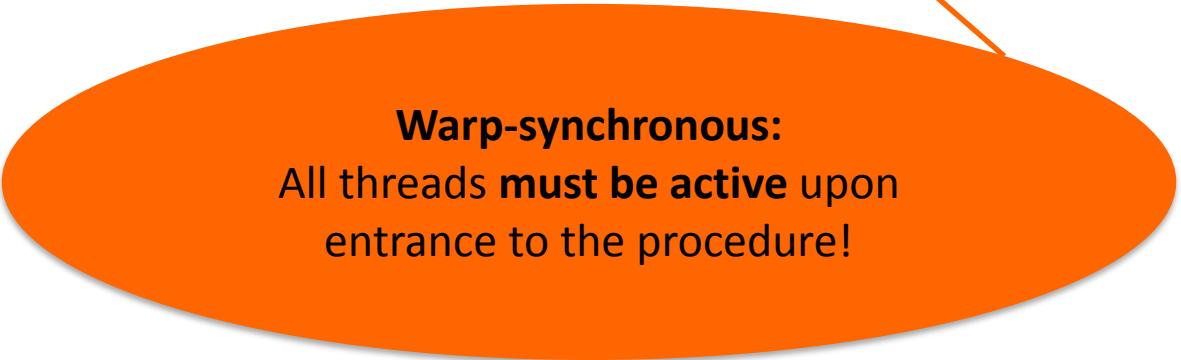




# Warp-Synchronous Programming

```
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.



**Warp-synchronous:**  
All threads **must be active** upon  
entrance to the procedure!



# Warp-Synchronous Programming

```
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);
        }
    }
}
```

Warp-synchronous wrapper for SIMD memory copy.



# Warp-Synchronous Programming

```
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.

Warp-level parallelism!

Mappings:





# Warp-Synchronous Programming

```
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.

Warp-level parallelism!

Mappings:

T<sub>0</sub> T<sub>1</sub> T<sub>2</sub> T<sub>3</sub>  
int value = [10 20 30 10]

increment

T<sub>0</sub> T<sub>1</sub> T<sub>2</sub> T<sub>3</sub>  
int value = [11 21 31 11]

Reductions:

T<sub>0</sub> T<sub>1</sub> T<sub>2</sub> T<sub>3</sub>  
int value = [10 20 30 10]

sum

T<sub>0</sub> T<sub>1</sub> T<sub>2</sub> T<sub>3</sub>  
int scalar = (70 70 70 70)



# 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 memory copy.

\* **Everywhere blocks** (early languages for SIMD machines):

- C\*
- MPL
- POMPC

Block wherein threads are  
**temporarily re-enabled!**

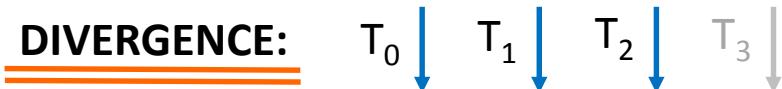


# Warp-Synchronous Programming: Everywhere blocks

```
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 memory copy.

\* **Everywhere blocks:**



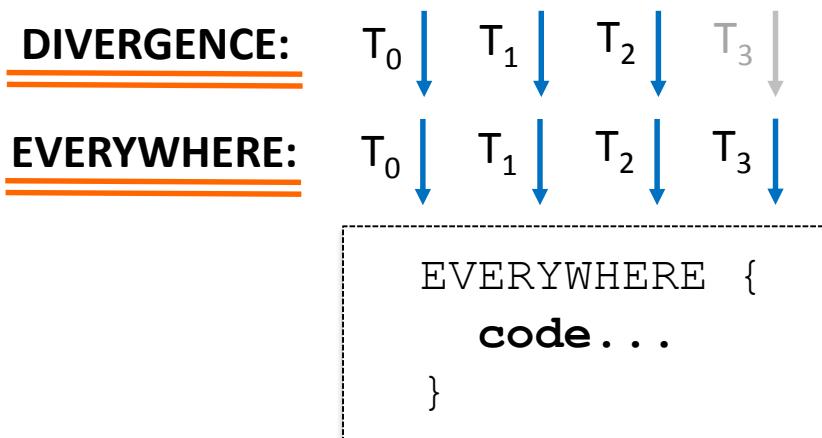


# Warp-Synchronous Programming: Everywhere blocks

```
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 memory copy.

\* **Everywhere blocks:**



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



# 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 memory copy.

\* Everywhere blocks:

DIVERGENCE: T<sub>0</sub> ↓ T<sub>1</sub> ↓ T<sub>2</sub> ↓ T<sub>3</sub> ↓

EVERYWHERE: T<sub>0</sub> ↓ T<sub>1</sub> ↓ T<sub>2</sub> ↓ T<sub>3</sub> ↓

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

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

DIVERGENCE: T<sub>0</sub> ↓ T<sub>1</sub> ↓ T<sub>2</sub> ↓ T<sub>3</sub> ↓

Divergences **restored**!



# Warp-Synchronous Programming: Everywhere blocks + Shuffle

```
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** (early languages for SIMD machines):

- C\*
- MPL
- POMPC

\* **Shuffle** (warp aware instruction):

shuffle(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 + Shuffle

```
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);
        }
    }
}
```

Warp-synchronous wrapper for SIMD memory copy.



# Warp-Synchronous Programming: Everywhere blocks + Shuffle

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

1. **everywhere** re-enables all threads!

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);  
        }  
    }  
}
```

Warp-synchronous wrapper for SIMD memory copy.



# Warp-Synchronous Programming: Everywhere blocks + Shuffle

```
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

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

```
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);  
        }  
    }  
}
```

Warp-synchronous wrapper for SIMD memory copy.



# Warp-Synchronous Programming: Everywhere blocks + Shuffle

```
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

1. **everywhere** re-enables all threads!
2. Skip formerly divergent threads!
3. Extracts values for current thread “i”.

```
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);  
        }  
    }  
}
```

Warp-synchronous wrapper for SIMD memory copy.



# Warp-Synchronous Programming: Everywhere blocks + Shuffle

```
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

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**.

```
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);  
        }  
    }  
}
```

Warp-synchronous wrapper for SIMD memory copy.



# Warp-Synchronous Programming: Everywhere blocks + Shuffle

```
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

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**.

```
void memcpy_wrapper(int **dest, int **src, int *N, int mask) {  
    #include "warp.h"  
    #include "warp_synchronous.h"  
  
    #define EVERYWHERE __attribute__((__warp_synchronous__(  
        warp_synchronous_everywhere,  
        warp_synchronous_no_shuffle,  
        warp_synchronous_no_shuffle)))  
  
    #define MEMCPY(dest, src, N) EVERYWHERE {  
        for (int i=0; i < N; i++) {  
            dest[i] = src[i];  
        }  
    }  
  
    MEMCPY(dest, src, N);  
}
```

The target architecture **must provide** a directive to **re-enable inactive threads**.

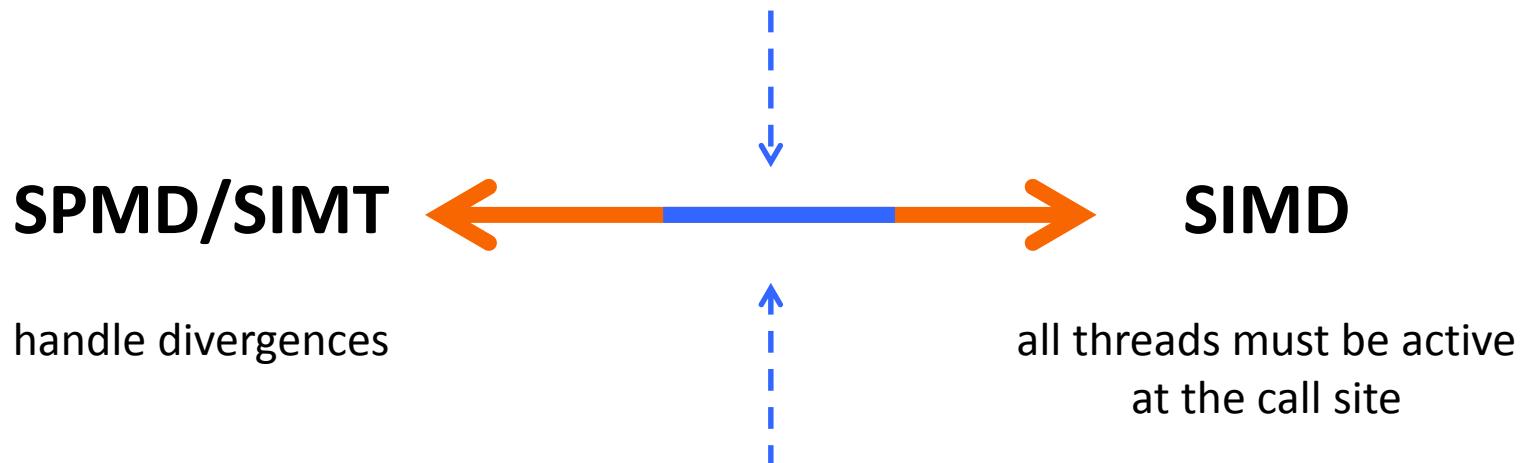
Warp-synchronous wrapper for SIMD memory copy.



# Warp-Synchronous Programming: Everywhere blocks + Shuffle

## **everywhere**

temporarily re-enables all threads within the warp

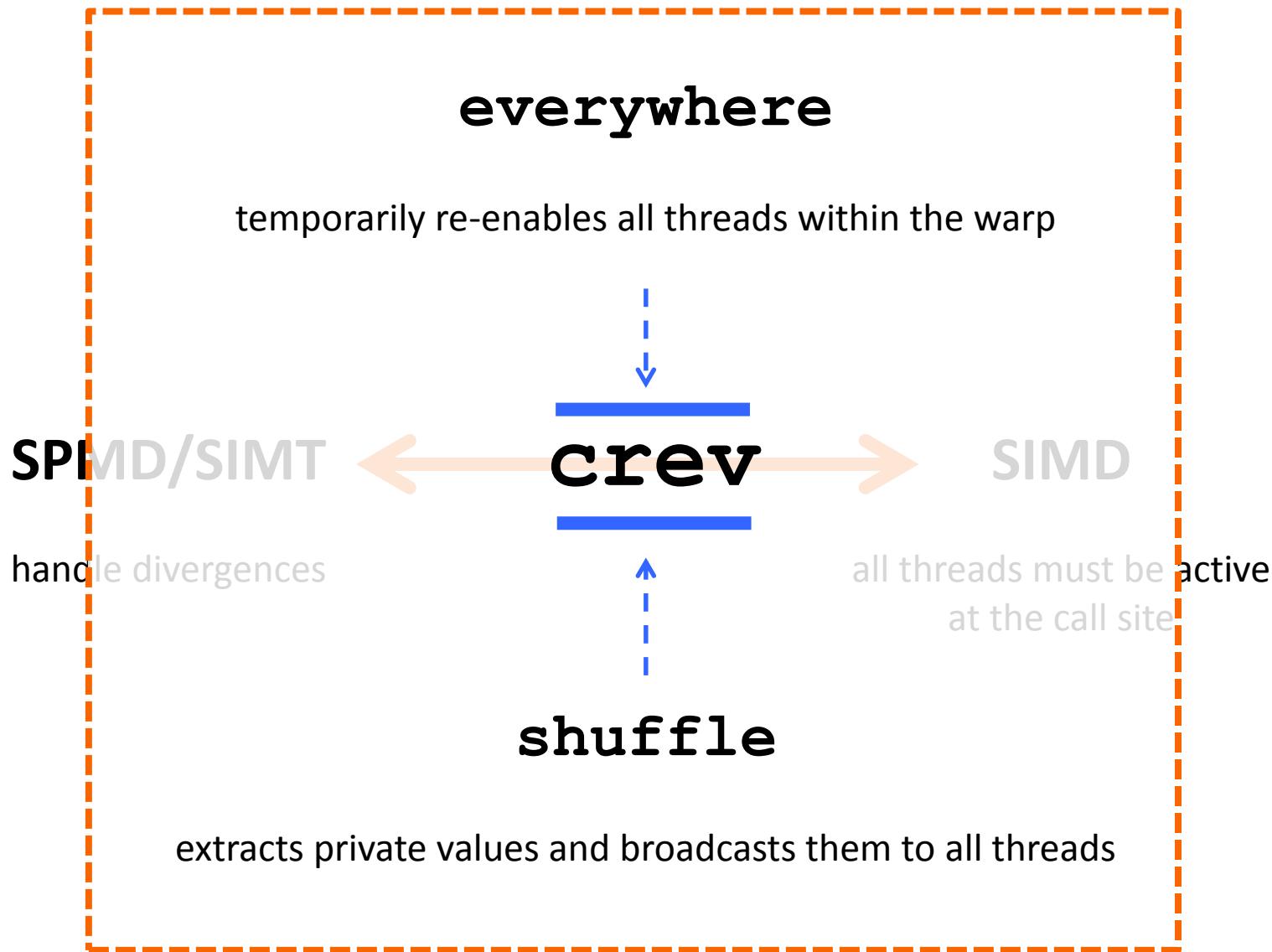


## **shuffle**

extracts private values and broadcasts them to all threads



# Warp-Synchronous Programming: Everywhere blocks + Shuffle





# Warp-Synchronous Programming: Everywhere blocks + Shuffle

We have defined the semantics of EVERYWHERE in the SIMD world:

|      |   |      |   |
|------|---|------|---|
| (Sp) | $\frac{P[pc] = \text{stop}}{(\Theta, \beta, \Sigma, \emptyset, \Lambda, P, pc) \rightarrow (\Theta, \beta, \Sigma)}$  | (Ss) | $\frac{P[pc] = \text{sync} \quad \Theta_n \neq \emptyset \quad (\Theta_n, \beta, \Sigma, (pc', \Theta_0, l, \emptyset) : \Pi, \Lambda, P, l) \rightarrow (\Theta', \beta', \Sigma')}{(\Theta, \beta, \Sigma, (pc', \emptyset, l, \Theta_n) : \Pi, \Lambda, P, pc) \rightarrow (\Theta', \beta', \Sigma')}$  |
| (Bt) | $\frac{\begin{array}{l} P[pc] = \text{bz } v, l \\ \text{split}(\Theta, \beta, v) = (\Theta, \emptyset) \end{array} \quad \text{push}(\Pi, \emptyset, pc, l) = \Pi' \quad (\Theta, \beta, \Sigma, \Pi', \Lambda, P, l) \rightarrow (\Theta', \beta', \Sigma')}{(\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc) \rightarrow (\Theta', \beta', \Sigma')}$        | (Sp) | $\frac{\begin{array}{l} P[pc] = \text{sync} \quad (\Theta_n, \beta, \Sigma, (\_, \emptyset, \_, \Theta_0) : \Pi, \Lambda, P, pc + 1) \rightarrow (\Theta', \beta', \Sigma') \\ (\Theta_0 \cup \Theta_n, \beta, \Sigma, \Pi, \Lambda, P, pc) \rightarrow (\Theta', \beta', \Sigma') \end{array}}{(\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc) \rightarrow (\Theta', \beta', \Sigma')}$   |
| (Bf) | $\frac{\begin{array}{l} P[pc] = \text{bz } v, l \\ \text{split}(\Theta, \beta, v) = (\emptyset, \Theta) \end{array} \quad \text{push}(\Pi, \emptyset, pc, l) = \Pi' \quad (\Theta, \beta, \Sigma, \Pi', \Lambda, P, pc + 1) \rightarrow (\Theta', \beta', \Sigma')}{(\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc) \rightarrow (\Theta', \beta', \Sigma')}$   | (Jp) | $\frac{P[pc] = \text{jump } l \quad (\Theta, \beta, \Sigma, \Pi, \Lambda, P, l) \rightarrow (\Theta', \beta', \Sigma')}{(\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc) \rightarrow (\Theta', \beta', \Sigma')}$   |
| (Bd) | $\frac{\begin{array}{l} P[pc] = \text{bz } v, l \\ \text{split}(\Theta, \beta, v) = (\Theta_0, \Theta_n) \end{array} \quad \text{push}(\Pi, \Theta_n, pc, l) = \Pi' \quad (\Theta_0, \beta, \Sigma, \Pi', \Lambda, P, pc + 1) \rightarrow (\Theta', \beta', \Sigma')}{(\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc) \rightarrow (\Theta', \beta', \Sigma')}$ | (Eb) | $\frac{P[pc] = \text{everywhere} \quad (\Theta_{\text{all}}, \beta, \Sigma, \emptyset, (\Theta, \Pi) : \Lambda, P, pc + 1) \rightarrow (\Theta', \beta', \Sigma')}{(\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc) \rightarrow (\Theta', \beta', \Sigma')}$  |
| (Ba) | $\frac{\begin{array}{l} P[pc] = \text{branch\_mask } T_{id}, l \\ T_{id} \in \Theta' \quad (\Theta, \beta, \Sigma, \Pi, (\Theta', \Pi') : \Lambda, P, l) \rightarrow (\Theta'', \beta', \Sigma') \end{array}}{(\Theta, \beta, \Sigma, \Pi, (\Theta', \Pi') : \Lambda, P, pc) \rightarrow (\Theta'', \beta', \Sigma')}$                                      | (Ee) | $\frac{P[pc] = \text{end\_everywhere} \quad (\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc + 1) \rightarrow (\Theta', \beta', \Sigma')}{(\_, \beta, \Sigma, \emptyset, (\Theta, \Pi) : \Lambda, P, pc) \rightarrow (\Theta', \beta', \Sigma')}$  |
| (Bi) | $\frac{\begin{array}{l} P[pc] = \text{branch\_mask } T_{id}, l \\ T_{id} \notin \Theta' \quad (\Theta, \beta, \Sigma, \Pi, (\Theta', \Pi') : \Lambda, P, pc + 1) \rightarrow (\Theta'', \beta', \Sigma') \end{array}}{(\Theta, \beta, \Sigma, \Pi, (\Theta', \Pi') : \Lambda, P, pc) \rightarrow (\Theta'', \beta', \Sigma')}$                              | (It) | $\frac{\begin{array}{l} P[pc] = \iota \\ \iota \notin \{\text{stop, bnz, bz, branch\_mask, sync, jump, everywhere, end\_everywhere}\} \\ (\Theta, \beta, \Sigma, \Theta_{\text{mask}}, \iota) \rightarrow (\beta', \Sigma') \end{array} \quad (\Theta, \beta', \Sigma', \Pi, (\Theta_{\text{mask}}, \Pi') : \Lambda, pc + 1) \rightarrow (\Theta', \beta'', \Sigma'')}{(\Theta, \beta, \Sigma, \Pi, (\Theta_{\text{mask}}, \Pi') : \Lambda, P, pc) \rightarrow (\Theta', \beta'', \Sigma'')}$ |

Semantics of everywhere in SIMD:  
encode the building blocks to implement this construct



# Warp-Synchronous Programming: Everywhere blocks + Shuffle

We have defined the semantics of **EVERYWHERE** in the SIMD world:

|      |   |      |   |
|------|---|------|---|
| (Sp) | $\frac{P[pc] = \text{stop}}{(\Theta, \beta, \Sigma, \emptyset, \Lambda, P, pc) \rightarrow (\Theta, \beta, \Sigma)}$  | (Ss) | $\frac{P[pc] = \text{sync} \quad \Theta_n \neq \emptyset \quad (\Theta_n, \beta, \Sigma, (pc', \Theta_0, l, \emptyset) : \Pi, \Lambda, P, l) \rightarrow (\Theta', \beta', \Sigma')} {(\Theta, \beta, \Sigma, (pc', \emptyset, l, \Theta_n) : \Pi, \Lambda, P, pc) \rightarrow (\Theta', \beta', \Sigma')}$   |
| (Bt) | $\frac{P[pc] = \text{bz } v, l \quad \text{split}(\Theta, \beta, v) = (\Theta, \emptyset)} {\text{push}(\Pi, \emptyset, pc, l) = \Pi'} \quad (\Theta, \beta, \Sigma, \Pi', \Lambda, P, l) \rightarrow (\Theta', \beta', \Sigma')$<br>$(\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc) \rightarrow (\Theta', \beta', \Sigma')$        | (Sp) | $\frac{P[pc] = \text{sync} \quad (\Theta_n, \beta, \Sigma, (\_, \emptyset, \_, \Theta_0) : \Pi, \Lambda, P, pc + 1) \rightarrow (\Theta', \beta', \Sigma')} {(\Theta_0 \cup \Theta_n, \beta, \Sigma, \Pi, \Lambda, P, pc) \rightarrow (\Theta', \beta', \Sigma')}$  |
| (Bf) | $\frac{P[pc] = \text{bz } v, l \quad \text{split}(\Theta, \beta, v) = (\emptyset, \Theta)} {\text{push}(\Pi, \emptyset, pc, l) = \Pi'} \quad (\Theta, \beta, \Sigma, \Pi', \Lambda, P, pc + 1) \rightarrow (\Theta', \beta', \Sigma')$<br>$(\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc) \rightarrow (\Theta', \beta', \Sigma')$   | (Jp) | $\frac{P[pc] = \text{jump } l \quad (\Theta, \beta, \Sigma, \Pi, \Lambda, P, l) \rightarrow (\Theta', \beta', \Sigma')} {(\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc) \rightarrow (\Theta', \beta', \Sigma')}$  |
| (Bd) | $\frac{P[pc] = \text{bz } v, l \quad \text{split}(\Theta, \beta, v) = (\Theta_0, \Theta_n)} {\text{push}(\Pi, \Theta_n, pc, l) = \Pi'} \quad (\Theta_0, \beta, \Sigma, \Pi', \Lambda, P, pc + 1) \rightarrow (\Theta', \beta', \Sigma')$<br>$(\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc) \rightarrow (\Theta', \beta', \Sigma')$ | (Eb) | $\frac{P[pc] = \text{everywhere} \quad (\Theta_{\text{all}}, \beta, \Sigma, \emptyset, (\Theta, \Pi) : \Lambda, P, pc + 1) \rightarrow (\Theta', \beta', \Sigma')} {(\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc) \rightarrow (\Theta', \beta', \Sigma')}$   |
| (Ba) | $\frac{\begin{array}{l} P[pc] = \text{branch\_mask } T_{id}, l \\ T_{id} \in \Theta' \end{array}} {(\Theta, \beta, \Sigma, \Pi, (\Theta', \Pi') : \Lambda, P, l) \rightarrow (\Theta'', \beta', \Sigma')}$<br>$(\Theta, \beta, \Sigma, \Pi, (\Theta', \Pi') : \Lambda, P, pc) \rightarrow (\Theta'', \beta', \Sigma')$            | (Ee) | $\frac{P[pc] = \text{end\_everywhere} \quad (\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc + 1) \rightarrow (\Theta', \beta', \Sigma')} {(\_, \beta, \Sigma, \emptyset, (\Theta, \Pi) : \Lambda, P, pc) \rightarrow (\Theta', \beta', \Sigma')}$   |
| (Bi) | $\frac{\begin{array}{l} P[pc] = \text{branch\_mask } T_{id}, l \\ T_{id} \notin \Theta' \end{array}} {(\Theta, \beta, \Sigma, \Pi, (\Theta', \Pi') : \Lambda, P, pc + 1) \rightarrow (\Theta'', \beta', \Sigma')}$<br>$(\Theta, \beta, \Sigma, \Pi, (\Theta', \Pi') : \Lambda, P, pc) \rightarrow (\Theta'', \beta', \Sigma')$    | (It) | $\frac{\begin{array}{l} P[pc] = \iota \\ \iota \notin \{\text{stop, bnz, bz, branch\_mask, sync, jump, everywhere, end\_everywhere}\} \\ (\Theta, \beta, \Sigma, \Theta_{\text{mask}}, \iota) \rightarrow (\beta', \Sigma') \end{array}} {(\Theta, \beta', \Sigma', \Pi, (\Theta_{\text{mask}}, \Pi') : \Lambda, pc + 1) \rightarrow (\Theta', \beta'', \Sigma'')}$<br>$(\Theta, \beta, \Sigma, \Pi, (\Theta_{\text{mask}}, \Pi') : \Lambda, P, pc) \rightarrow (\Theta', \beta'', \Sigma'')$ |



**SWI Prolog**

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**.

# Warp-Synchronous Programming: CREV

**crev** memcmp (i)

| Instructions   |                       |  |
|----------------|-----------------------|--|
|                | v0 = $\downarrow$ tid |  |
|                | v1 = ( $v_0 == 0$ )   |  |
| bz v1, Done    |                       |  |
|                | v2 = $4 * (tid + 1)$  |  |
| everywhere     |                       |  |
|                | v8 = 0                |  |
| Loop           | jmp_mask v8, Call     |  |
|                | jump Next             |  |
| Call           | v3 = shfl(v2, v8)     |  |
|                | v4 = v3 + tid         |  |
|                | v5 = $\downarrow$ v4  |  |
|                | v6 = v5 + 1           |  |
|                | $\uparrow$ v4 = v6    |  |
| Next           | v8 = v8 + 1           |  |
|                | v7 = ( $v_8 == 4$ )   |  |
|                | bz v7, Loop           |  |
| end everywhere |                       |  |
| Done           | tid = 1               |  |
|                | sync                  |  |

Function declaration:  $f(p_1, \dots, p_n);$

Function call:  $f(t_1 a_1, \dots, t_n a_n);$

**Function extract( $t^n, p^n, a^n, i$ )**

```
for  $k \in 1 \dots n$  do
    if  $t_k == uniform$  then
         $p_k = a_k;$ 
    if  $t_k == varying$  then
        shfl( $a_k, i$ );
```

```

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  $\neq W$ ) loop
9      end_everywhere        ; ; end CREV
```



DEPARTMENT OF COMPUTER SCIENCE  
UNIVERSIDADE FEDERAL DE MINAS GERAIS  
FEDERAL UNIVERSITY OF MINAS GERAIS, BRAZIL

# FUNCTION CALL RE-VECTORIZATION



# 8 Function Call Re-Vectorization: *Reprise*

```
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.

SIMD function

# 8 Function Call Re-Vectorization: *Reprise*

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

SIMD function

SIMD implementation of memory copy.

```
void memcpy_wrapper(int **dest, int **src, int *N, int mask) {  
    memcpy<<<1, 4>>>(dest[tid], src[tid], N[tid]);  
}
```

Too much  
overhead

CUDA's *nested kernel call: Dynamic parallelism*

# 8 Function Call Re-Vectorization: *Reprise*

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

SIMD function

SIMD implementation of memory copy.

```
void memcpy_wrapper(int **dest, int **src, int *N, int mask) {  
    memcpy<<<1, 4>>>(dest[tid], src[tid], N[tid]);  
}
```

Too much overhead

CUDA's nested kernel call: Dynamic parallelism

```
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);  
        }  
    }  
}
```

Warp-synchronous wrapper for SIMD memory copy.

Too many lines of code

# 8 Function Call Re-Vectorization: *Reprise*

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

Simplicity + Performance, *a.k.a.*  
**CREV**

SIMD implementation of memory copy.

```
void memcpy_wrapper(int **dest, int **src, int *N, int mask) {  
    memcpy<<<1, 4>>>(dest[tid], src[tid], N[tid]);  
}
```

CUDA's *nested kernel call: Dynamic parallelism*

```
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);  
        }  
    }  
}
```

Warp-synchronous wrapper for SIMD memory copy.

Too much  
overhead

Too many  
lines of code



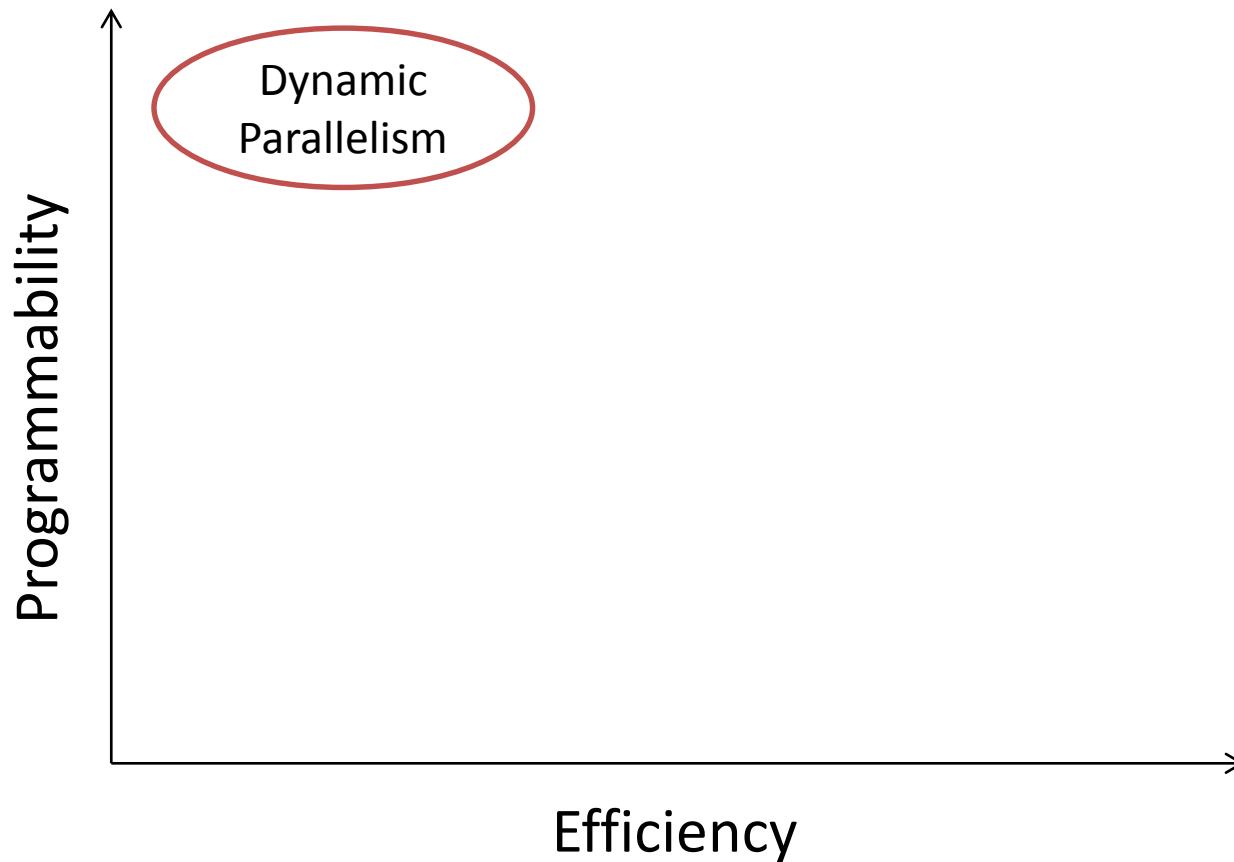
# Function Call Re-Vectorization: *Reprise*

---



# 8 Function Call Re-Vectorization: *Reprise*

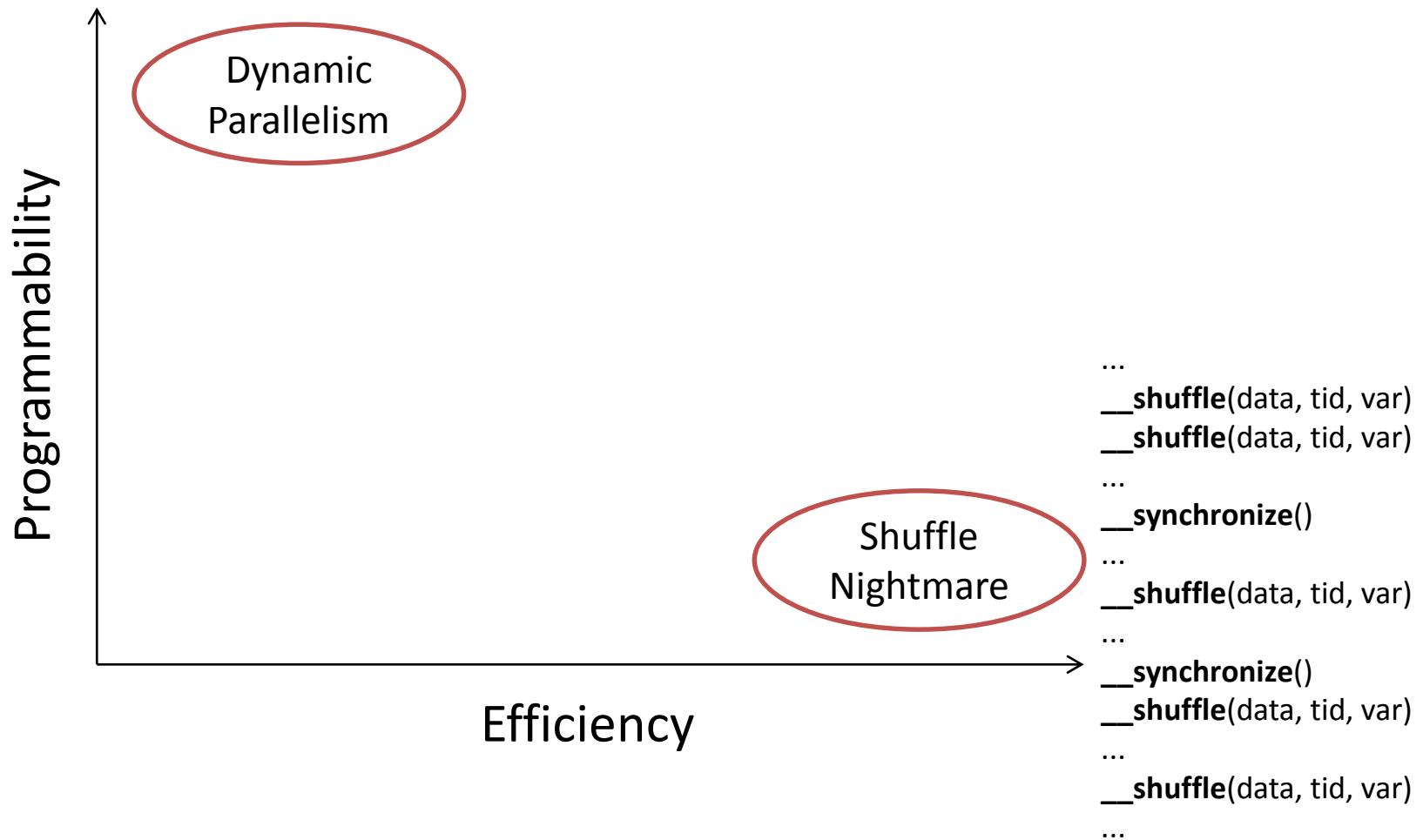
CUDA: kernel<<<#warps, #threads>>>(args...)





# Function Call Re-Vectorization: *Reprise*

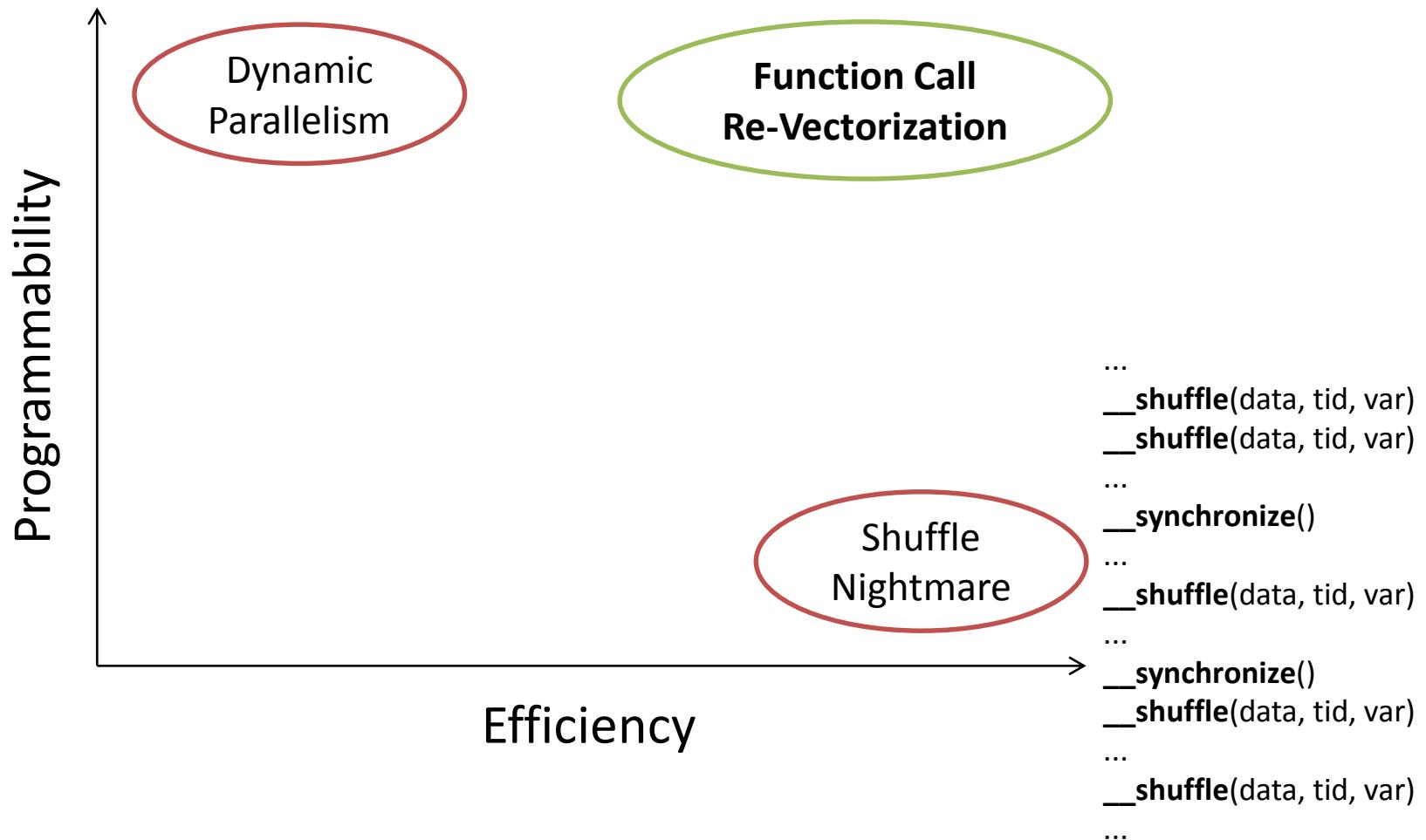
CUDA: kernel<<<#warps, #threads>>>(args...)





# Function Call Re-Vectorization: *Reprise*

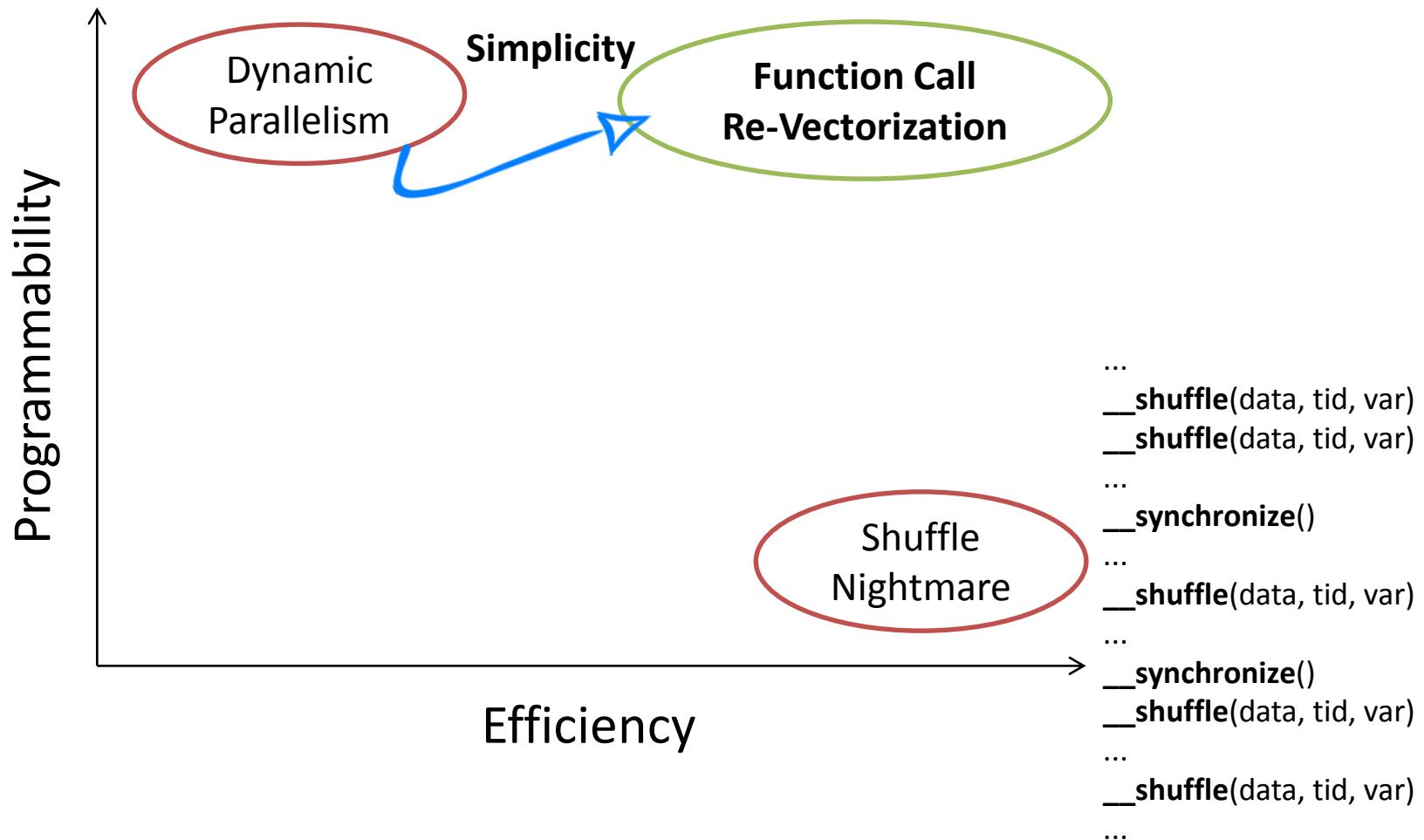
CUDA: kernel<<<#warps, #threads>>>(args...)





# Function Call Re-Vectorization: *Reprise*

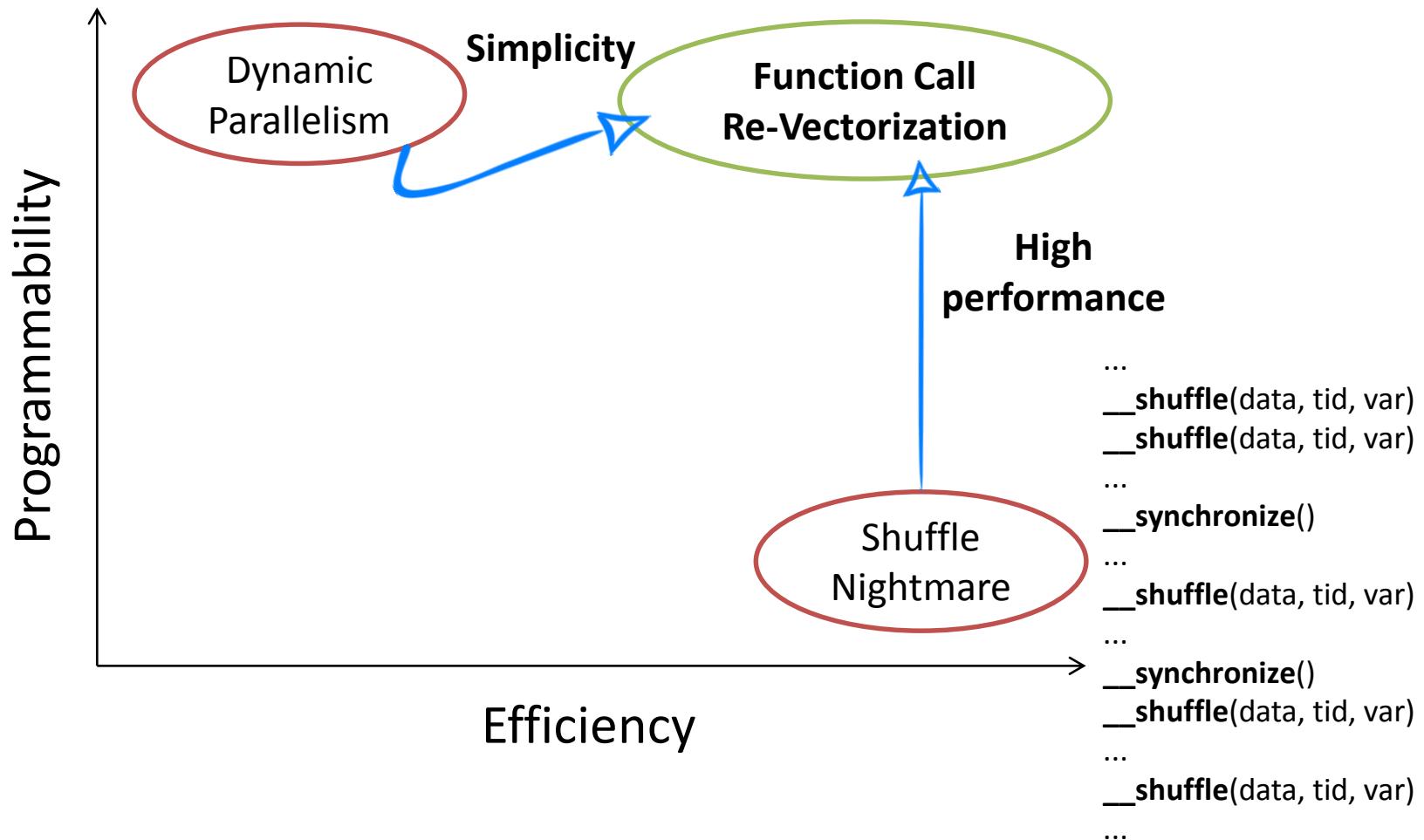
CUDA: kernel<<<#warps, #threads>>>(args...)





# Function Call Re-Vectorization: *Reprise*

CUDA: kernel<<<#warps, #threads>>>(args...)

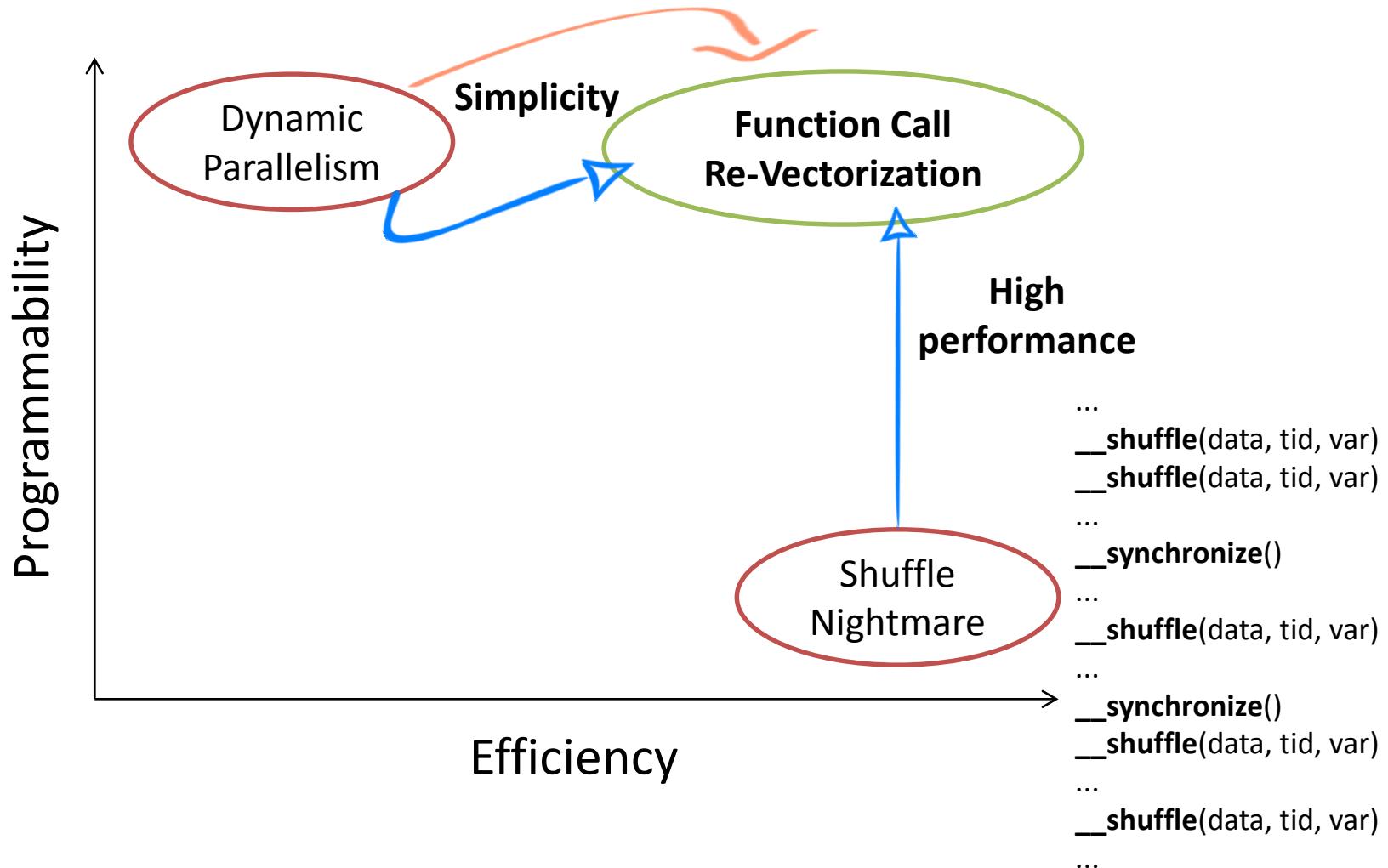




# Function Call Re-Vectorization: *Reprise*

CUDA: kernel<<<#warps, #threads>>>(args...)

**Re-enable all threads within warp,  
avoiding kernel allocation and scheduling**

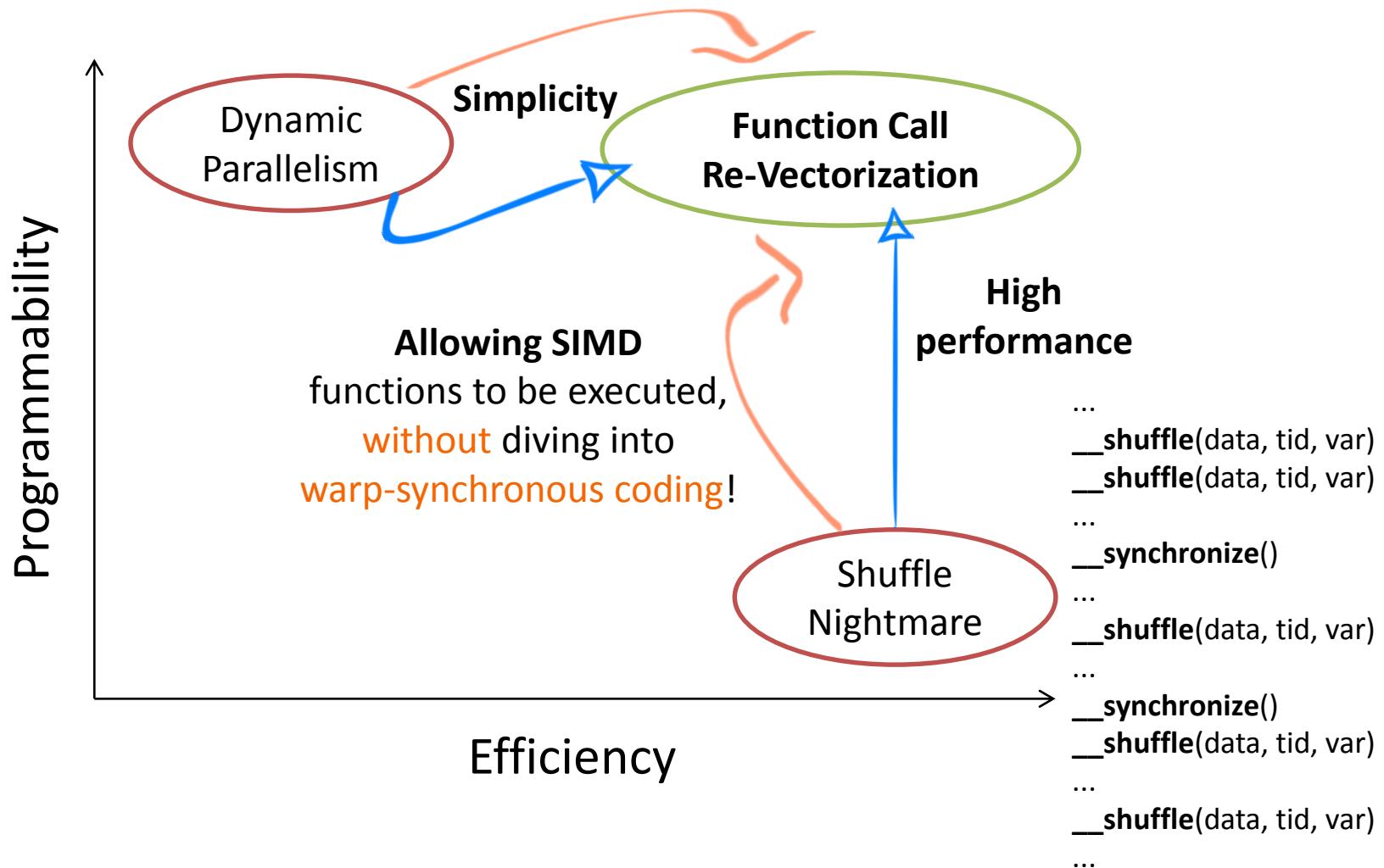




# Function Call Re-Vectorization: *Reprise*

CUDA: kernel<<<#warps, #threads>>>(args...)

**Re-enable all threads within warp,  
avoiding kernel allocation and scheduling**



# 8 Function Call Re-Vectorization: Properties

## Composability

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

```
// Traverses the matrix in a depth-first fashion
void dfs(uniform struct Graph& graph, uniform int root, float * uniform f) {

    if (graph.node[root].visited) return;
    graph.node[root].visited = true;

    // Eventual computations
    f[root] = graph.node[root].length /
        (float) graph.num_nodes;

    // Traversal
    foreach (i = 0 ... graph.node[root].length) {

        int child = graph.node[root].edge[i].node;
        if (!graph.node[child].visited) {
            crev dfs(graph, child, f);
        }
    }
}
```

**Important benefits** when new work is invoked within an executing GPU program include **removing the burden on the programmer to marshal and transfer the data** on which to operate. **Additional parallelism can be exposed** to the GPU's hardware schedulers and load balancers dynamically, **adapting in response to data-driven decisions** or workloads. **Algorithms and programming patterns** that had previously required modifications to eliminate recursion, irregular loop structure, or other constructs that do not fit a flat, single-level of parallelism **can be more transparently expressed**.

Dynamic Parallelism in CUDA

Source:[http://developer.download.nvidia.com/assets/cuda/files/CUDA\\_Downloads/TechBrief\\_Dynamic\\_Parallelism\\_in\\_CUDA.pdf](http://developer.download.nvidia.com/assets/cuda/files/CUDA_Downloads/TechBrief_Dynamic_Parallelism_in_CUDA.pdf)

# 8 Function Call Re-Vectorization: Properties

## 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.

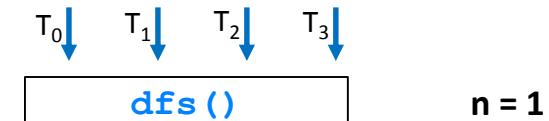
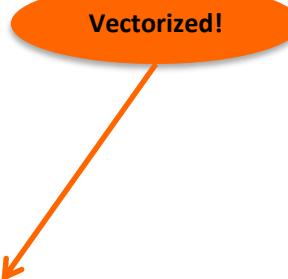
If the call is recursive, up to **W<sup>N</sup>** times.

```
// Traverses the matrix in a depth-first fashion
void dfs(uniform struct Graph& graph, uniform int root,
    float * uniform f) {
    if (graph.node[root].visited) return;
    graph.node[root].visited = true;

    // Eventual computations
    f[root] = graph.node[root].length /
        (float) graph.num_nodes;

    // Traversal
    foreach (i = 0 ... graph.node[root].length) {

        int child = graph.node[root].edge[i].node;
        if (!graph.node[child].visited) {
            crev dfs(graph, child, f);
        }
    }
}
```



# 8 Function Call Re-Vectorization: Properties

## 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.

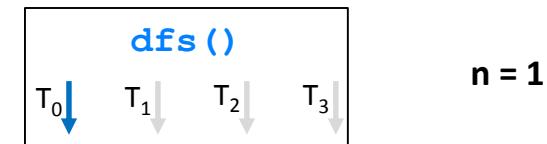
If the call is recursive, up to **W<sup>N</sup>** times.

```
// Traverses the matrix in a depth-first fashion
void dfs(uniform struct Graph& graph, uniform int root,
         float * uniform f) {
    if (graph.node[root].visited) return;
    graph.node[root].visited = true;

    // Eventual computations
    f[root] = graph.node[root].length /
        (float) graph.num_nodes;

    // Traversal
    foreach (i = 0 ... graph.node[root].length) {

        int child = graph.node[root].edge[i].node;
        if (!graph.node[child].visited) {
            crev dfs(graph, child, f);
        }
    }
}
```



# 8 Function Call Re-Vectorization: Properties

## 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.

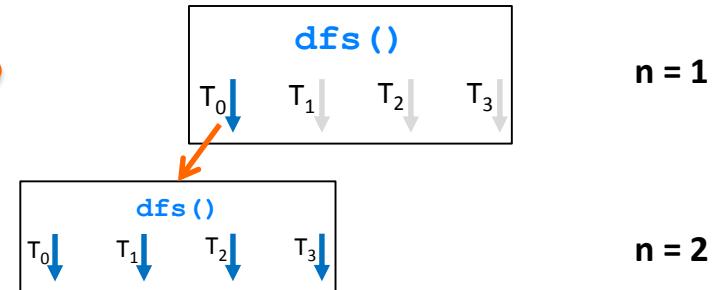
If the call is recursive, up to **W<sup>N</sup>** times.

```
// Traverses the matrix in a depth-first fashion
void dfs(uniform struct Graph& graph, uniform int root,
         float * uniform f) {
    if (graph.node[root].visited) return;
    graph.node[root].visited = true;

    // Eventual computations
    f[root] = graph.node[root].length /
        (float) graph.num_nodes;

    // Traversal
    foreach (i = 0 ... graph.node[root].length) {

        int child = graph.node[root].edge[i].node;
        if (!graph.node[child].visited) {
            crev dfs(graph, child, f);
        }
    }
}
```



# 8 Function Call Re-Vectorization: Properties

## Multiplicative composition

The target **crev** function runs once per active thread.

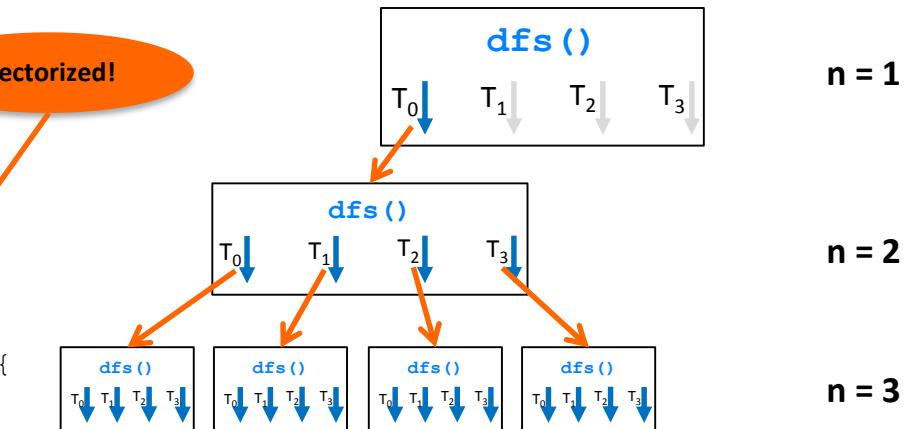
In a warp of **W** threads, the function may run up to **W** times.

If the call is recursive, up to **W<sup>N</sup>** times.

```
// Traverses the matrix in a depth-first fashion
void dfs(uniform struct Graph& graph, uniform int root,
         float * uniform f) {
    if (graph.node[root].visited) return;
    graph.node[root].visited = true;

    // Eventual computations
    f[root] = graph.node[root].length /
        (float) graph.num_nodes;

    // Traversal
    foreach (i = 0 ... graph.node[root].length) {
        int child = graph.node[root].edge[i].node;
        if (!graph.node[child].visited) {
            crev dfs(graph, child, f);
        }
    }
}
```



# 8 Function Call Re-Vectorization: Properties

## Multiplicative composition

The target **crev** function runs once per active thread.

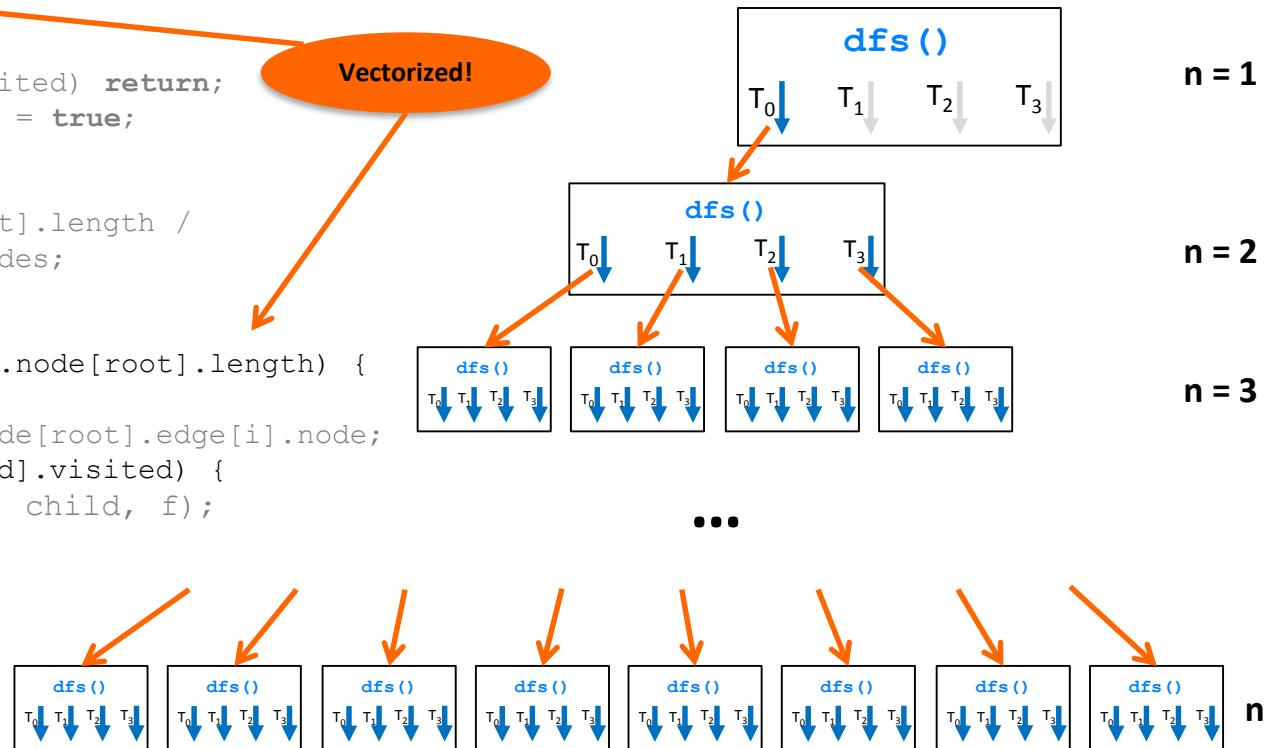
In a warp of **W** threads, the function may run up to **W** times.

If the call is recursive, up to **W<sup>N</sup>** times.

```
// Traverses the matrix in a depth-first fashion
void dfs(uniform struct Graph& graph, uniform int root,
         float * uniform f) {
    if (graph.node[root].visited) return;
    graph.node[root].visited = true;

    // Eventual computations
    f[root] = graph.node[root].length /
        (float) graph.num_nodes;

    // Traversal
    foreach (i = 0 ... graph.node[root].length) {
        int child = graph.node[root].edge[i].node;
        if (!graph.node[child].visited) {
            crev dfs(graph, child, f);
        }
    }
}
```



# 8 Function Call Re-Vectorization: Properties

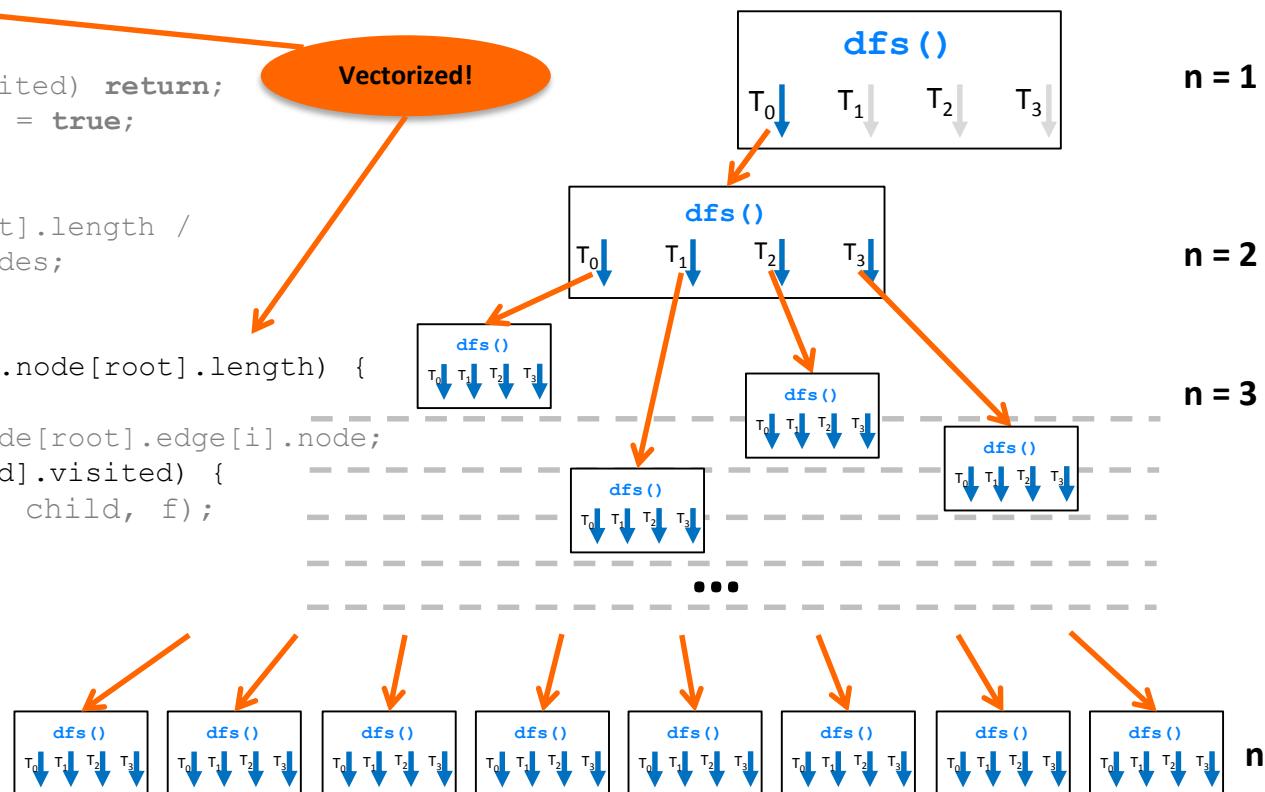
## Commutativity

There is **no predefined order** between execution of `crev's` target function.

```
// Traverses the matrix in a depth-first fashion
void dfs(uniform struct Graph& graph, uniform int root,
         float * uniform f) {
    if (graph.node[root].visited) return;
    graph.node[root].visited = true;

    // Eventual computations
    f[root] = graph.node[root].length /
        (float) graph.num_nodes;

    // Traversal
    foreach (i = 0 ... graph.node[root].length) {
        int child = graph.node[root].edge[i].node;
        if (!graph.node[child].visited) {
            crev dfs(graph, child, f);
        }
    }
}
```



# 8 Function Call Re-Vectorization: Properties

## Synchronization parity

Synchronization primitives remain correct, regardless of the **crev** nested level.  
**crev** uses a **context stack** to keep track of divergences.

```
...  
__shuffle(data, tid, var)  
__shuffle(data, tid, var)  
...  
__synchronize()  
...  
__shuffle(data, tid, var)  
...  
__synchronize()  
__shuffle(data, tid, var)  
...  
__shuffle(data, tid, var)  
...
```



===== : **crev** =====>

```
...  
__shuffle(data, tid, var)  
__shuffle(data, tid, var)  
...  
__synchronize()  
...  
crev f()  
...  
__shuffle(data, tid, var)  
...  
__synchronize()  
__shuffle(data, tid, var)  
...  
__shuffle(data, tid, var)  
...
```





DEPARTMENT OF COMPUTER SCIENCE  
UNIVERSIDADE FEDERAL DE MINAS GERAIS  
FEDERAL UNIVERSITY OF MINAS GERAIS, BRAZIL

## EXPERIMENTAL EVALUATION

---



# 8 Experimental Evaluation: String Matching

```
string T = text, P = pattern;

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);
}
```

We vectorize both loops

This is a CREV call!

**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.

# Experimental Evaluation: String Matching

```
string T = text, P = pattern;

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);
}
```

SIMD function:  
all threads must be active

This is a  
CREV call!

**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.

# 8 Experimental Evaluation: String Matching

```
string T = text, P = pattern;

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);
}

void NaiveStringMatch() {
    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);
    }
}
```

**SIMD function:**  
all threads must be active

This is a  
**CREV** call!

Naïve parallel approach



# Experimental Evaluation: Environment 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:

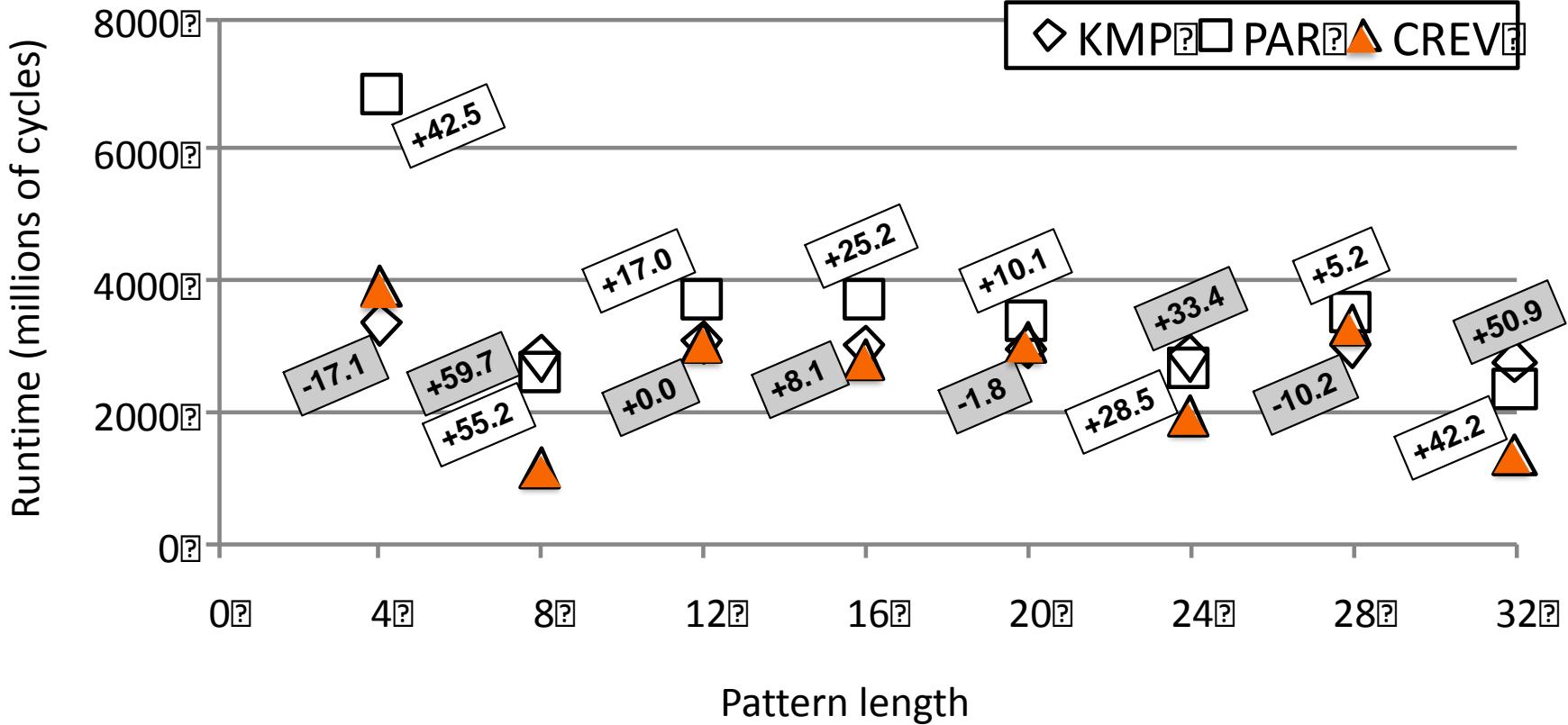
- **CREV**: our contribution
- **PAR**: implementation based on ISPC's constructs
- **SEQ**: state-of-the-art sequential implementation
- **Launch**: implementation using **dynamic parallelism** (pthreads)

## Environment

- 6-core 2.00 GHz Intel Xeon E5-2620 CPU (8-wide AVX vector units)
- Linux Ubuntu 12.04 3.2.0



# Experimental Evaluation: String Matching



Numbers show percentage of speedup of CREV over PAR (white) and KMP (grey)  
**Input:** 256MB in 5M lines from books from Project Gutenberg



# Function Call Re-Vectorization: CREV



Execution times (in millions of cycles):

★ Fastest; ⚜ 1<sup>st</sup> runner up; ⚪ 2<sup>nd</sup> runner up.

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

## 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 5bi 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].



# Function Call Re-Vectorization: CREV



Execution times (in millions of cycles):

★ Fastest; ⚜ 1<sup>st</sup> runner up; ⚪ 2<sup>nd</sup> runner up.

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

## Datasets:

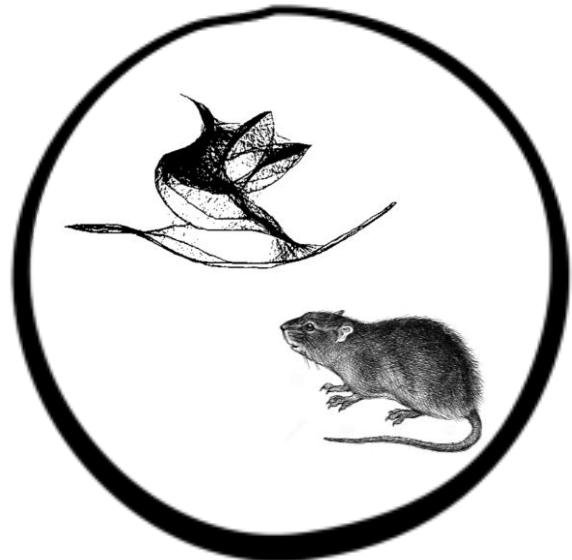
- **bin-L20K-P16:** 10K strings of 0s and 1s, each of length 20K, and target pattern of length 16.
- **txt-256MB-P16:** 256MB in 5bi lines from books from Project Gutenberg; target pattern has length 16.
- **erdos-renyi:** random Erdos-Renyi graph with 2048 nodes and 80% probability of edges.
- **octree-D5:**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



DEPARTMENT OF COMPUTER SCIENCE  
UNIVERSIDADE FEDERAL DE MINAS GERAIS  
FEDERAL UNIVERSITY OF MINAS GERAIS, BRAZIL

# CONTRIBUTIONS

---



# Rat Contributions: Rat



## Inference of Peak Density of Indirect Branches to Detect ROP Attacks

Márcio Tymorbi | Robson E. A. Moreira | Fernando Magno Quettmeier Pessin  
Department of Computer Science,  
UFMG Brazil  
{matto@dcc.ufmg.br}|{robson}@dcc.ufmg.br|{fmquettmeier}@dcc.ufmg.br

**Abstract**  
A program subject to Return-Oriented Programming (ROP) attacks can be detected by monitoring the density of indirect branches. From this observation, several techniques have been proposed to detect ROP attacks. These techniques are mostly based on the analysis of indirect branches in the trace of a target application. These techniques are attack specific and are not necessarily the same for every application. This paper proposes a more general approach to detect ROP attacks. As an alternative, we introduce a new method to estimate the maximum density of indirect branches possible for a given program. This method is based on the analysis of the execution trace of the program. The main idea is to provide a way to detect ROP-based attacks with some level of confidence. We evaluate our approach in two different applications: a compiler and a web browser. Our results show that our approach is able to detect ROP attacks with a reasonable reliability, even when it is applied to different applications.

**Keywords** Return-Oriented Programming, ROP, static analysis, compiler, web browser.

**General Terms** Languages, Security, Experimentation, Software Engineering, System, Program Analysis, Security.

**1. Introduction**  
Return-Oriented Programming (ROP) attacks are an important threat to security. By tracking the density of indirect branches present in the execution trace of a target application, it is possible to detect ROP-based attacks with some level of confidence. Several techniques have been proposed to detect ROP attacks independently determined by Cather et al. [2] and Galan et al. [3]. These techniques are attack specific and are not necessarily the same for every application. This paper proposes a more general approach to detect ROP attacks. As an alternative, we introduce a new method to estimate the maximum density of indirect branches possible for a given program. This method is based on the analysis of the execution trace of the program. The main idea is to provide a way to detect ROP-based attacks with some level of confidence. We evaluate our approach in two different applications: a compiler and a web browser. Our results show that our approach is able to detect ROP attacks with a reasonable reliability, even when it is applied to different applications.

<http://www.dcc.ufmg.br/~matto/paper.pdf>

## Paper published in CGO'16

### Inference of Peak Density of Indirect Branches to Detect ROP Attacks



**ACM CGO-SRC'16 winner  
(1<sup>st</sup> place: golden medal)**

## RIP-ROP Deducer

### Webpage with static analysis available

<http://cuda.dcc.ufmg.br/rip-rop-deducer>



matto@dcc.ufmg.br  
roberto@dcc.ufmg.br  
fmatto@dcc.ufmg.br



### Artifact Evaluated paper benchmarks

XV Sistemas Realizados em Engenharia de Sistemas Computacionais — SBSEG’15

## Inferência Estática da Frequência Máxima de Instruções de Retorno para Detecção de Ataques ROP

Roberto Emilio, Márcio Tymorbi e Fernando Magno Quettmeier Pessin  
Universidade Federal de Minas Gerais  
{roberto, matto, fmquettmeier}@dcc.ufmg.br

**Abstract.** Um programa subjetivo a Return Oriented Programming (ROP) utiliza ROP quando sua execução trace com alta frequência de retorno, indicando que seu objetivo é detectar ROP attacks. Essas técnicas são geralmente baseadas no estudo das instruções de retorno para detectar ROP attacks. Essas técnicas são consideradas serem iguais para todos os aplicativos. Esta pesquisa mostra que universalmente é possível estimar a densidade máxima de retorno necessária em diferentes aplicações.

**Resumo.** Programas que sofrem ataques baseados em Programação Orientada por Retorno (ROP) podem apresentar tracés de execução com alta densidade de retorno, indicando que seu objetivo é detectar ROP attacks. Essas técnicas são geralmente baseadas no estudo das instruções de retorno para detectar ROP attacks. Essas técnicas são consideradas serem iguais para todos os aplicativos. Esta pesquisa mostra que universalmente é possível estimar a densidade máxima de retorno necessária em diferentes aplicações.

**1. Introdução**  
Ataques de tipo ROP (do inglês Return-Oriented Programming) estão entre os mais difíceis de detectar e prevenir. Inicialmente esses tipos ocorrem quando o usuário compõe mensagens para que o sistema execute códigos que ele não controla. Isso pode ser feito através de pacotes formados de dentro de ataques baseados na monitorização das freqüências de retorno de uma aplicação. Porém, existem muitos tipos de ataques ROP que não possuem a mesma densidade de retorno necessária em diferentes aplicações. Neste artigo apresentamos um algoritmo que estima estaticamente a maior densidade de retorno necessária para detectar ataques ROP em diferentes aplicações.

**Este trabalho é dedicado em homenagem ao professor Fernando Magno Quettmeier Pessin**

**E-mail:** matto@dcc.ufmg.br

**Editor:** Prof. Dr. Henrique Góes

**Editor:** Prof. Dr. Henrique



# Contributions: Swan



## function call revectorization

# Paper published in PPoPP'17

## Function Call Re-Vectorization



# **Artifact Evaluated paper benchmarks**



# SWI Prolog

# Semantics of everywhere blocks in SIMD context

$\mu$ SIMD Prolog abstract machine



## Definição Semântica de Blocos Everywhere para Programação SIMD

Rubens Emilia Alves Martins<sup>1</sup>, Sylvain Collange<sup>2</sup> e Fláviausko Magno Quintão Pereira<sup>3</sup>

<sup>1</sup> UFMG - Arnaldo Antônio Carlos, 6627, 31.270-010, Belo Horizonte, MG, Brasil  
e-mail: fernanda@decc.ufmg.br

<sup>2</sup> INRIA — Centre de recherche Rennes, Bretagne Atlantique, Campus de Beaulieu,  
35012 RENNES, France e-mail: clemente.inria.fr

# Paper published in SBLP'16

## Definição Semântica de Blocos Everywhere para Programação SIMD

mentados sobre el. Tal información se encuentra hoy disponible para los países que la implementan. Pueden ser útiles para identificar otros tipos de construcciones no mencionadas.

<sup>1</sup>Fonte: [www.bcb.gov.br](http://www.bcb.gov.br). Atualizado: 14/07/2011.

# QUESTIONS?

**Email us:**

Rubens Emilio Alves Moreira [[rubens@dcc.ufmg.br](mailto:rubens@dcc.ufmg.br)]

Fernando Magno Quintão Pereira [[fernando@dcc.ufmg.br](mailto:fernando@dcc.ufmg.br)]

**Check our websites:**

<http://cuda.dcc.ufmg.br/swan>

<http://cuda.dcc.ufmg.br/rip-rop-deducer>

*Everything can happen.  
Everything is possible and probable.  
Time and space do not exist.  
On a flimsy framework of reality,  
the imagination spins,  
weaving new patterns.*

Ingmar Bergman – Fanny and Alexander

