Compilers Laboratory

DCC
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

ICEx
Instituto de Ciências Exatas

UFMG

Function Call Re-Vectorization

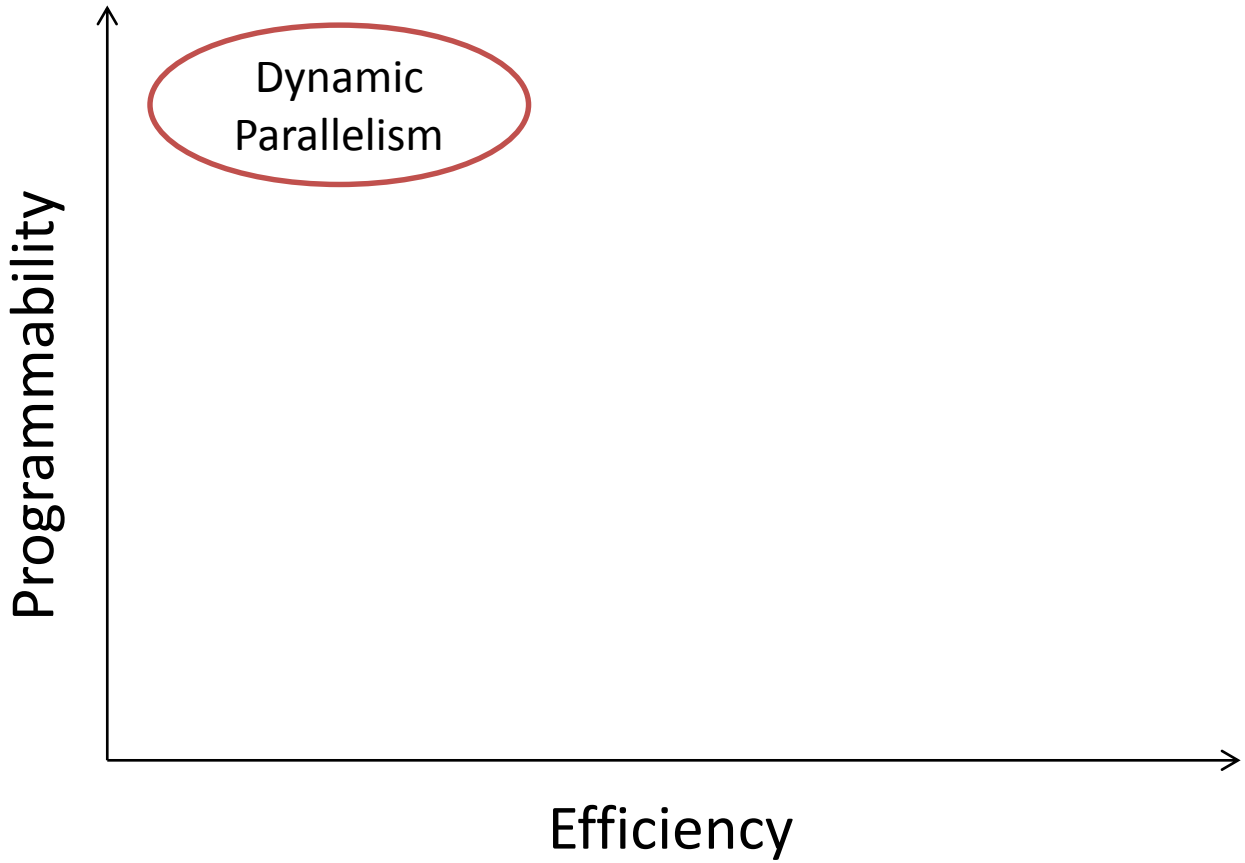
Pupil: Rubens Emilio Alves Moreira

Advisor: Fernando Magno Quintão Pereira

Function Call Re-Vectorization

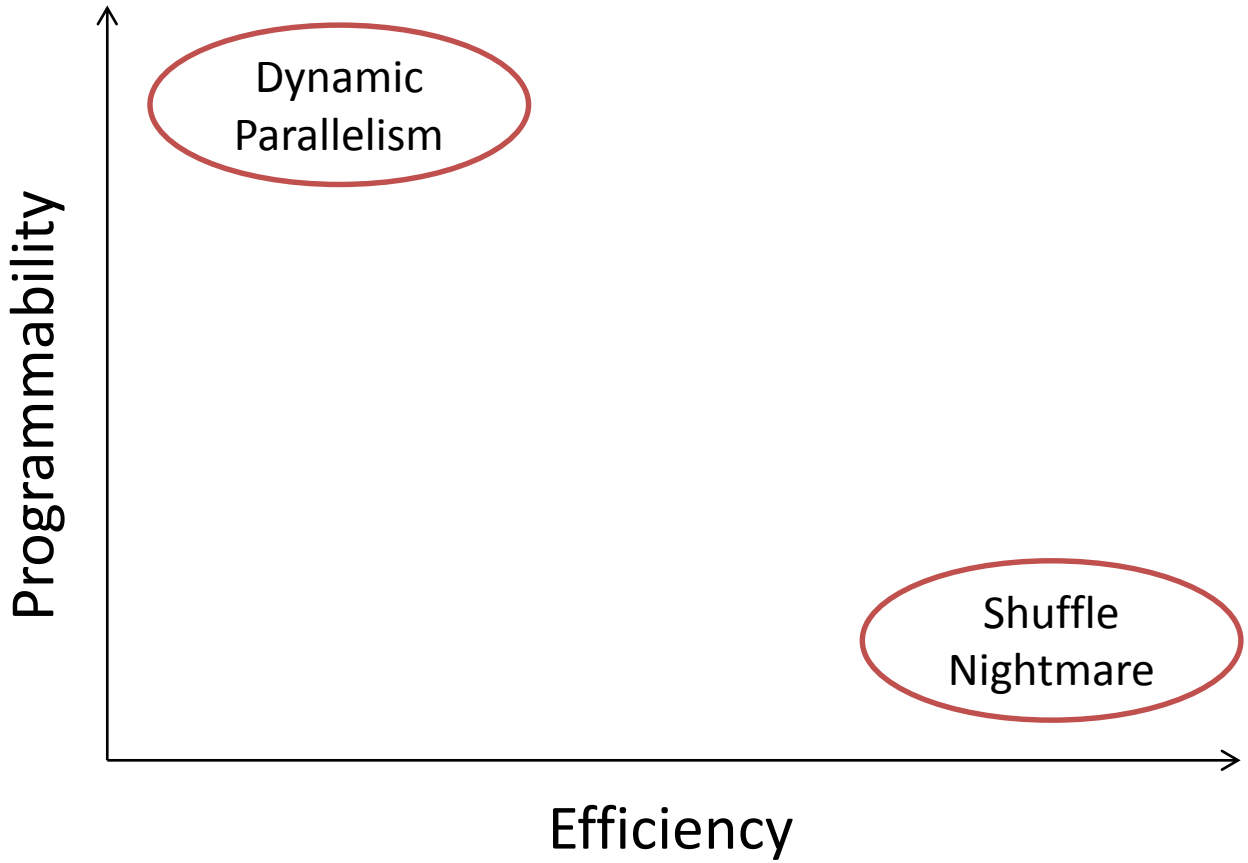
Function Call Re-Vectorization

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



Function Call Re-Vectorization

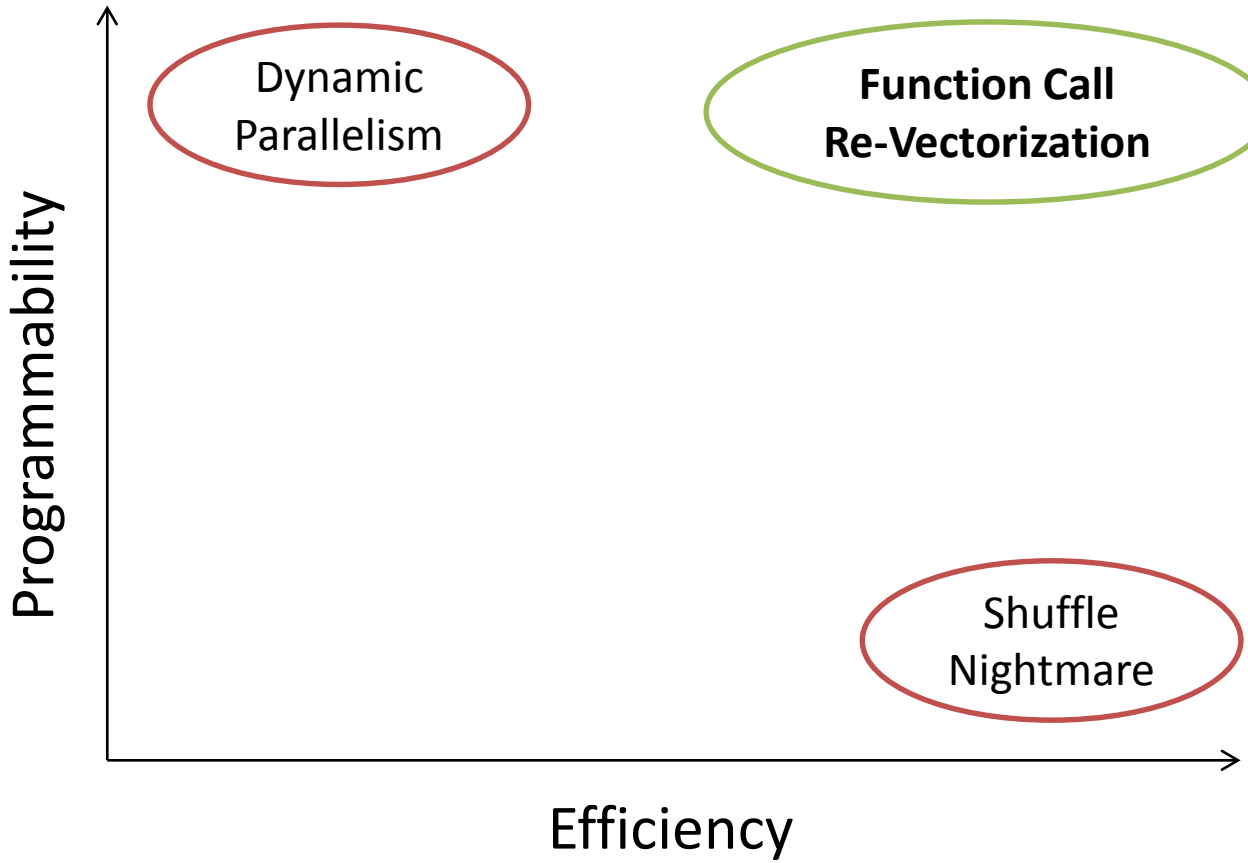
CUDA: kernel**<<<#warps, #threads>>>**(args...)



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

Function Call Re-Vectorization

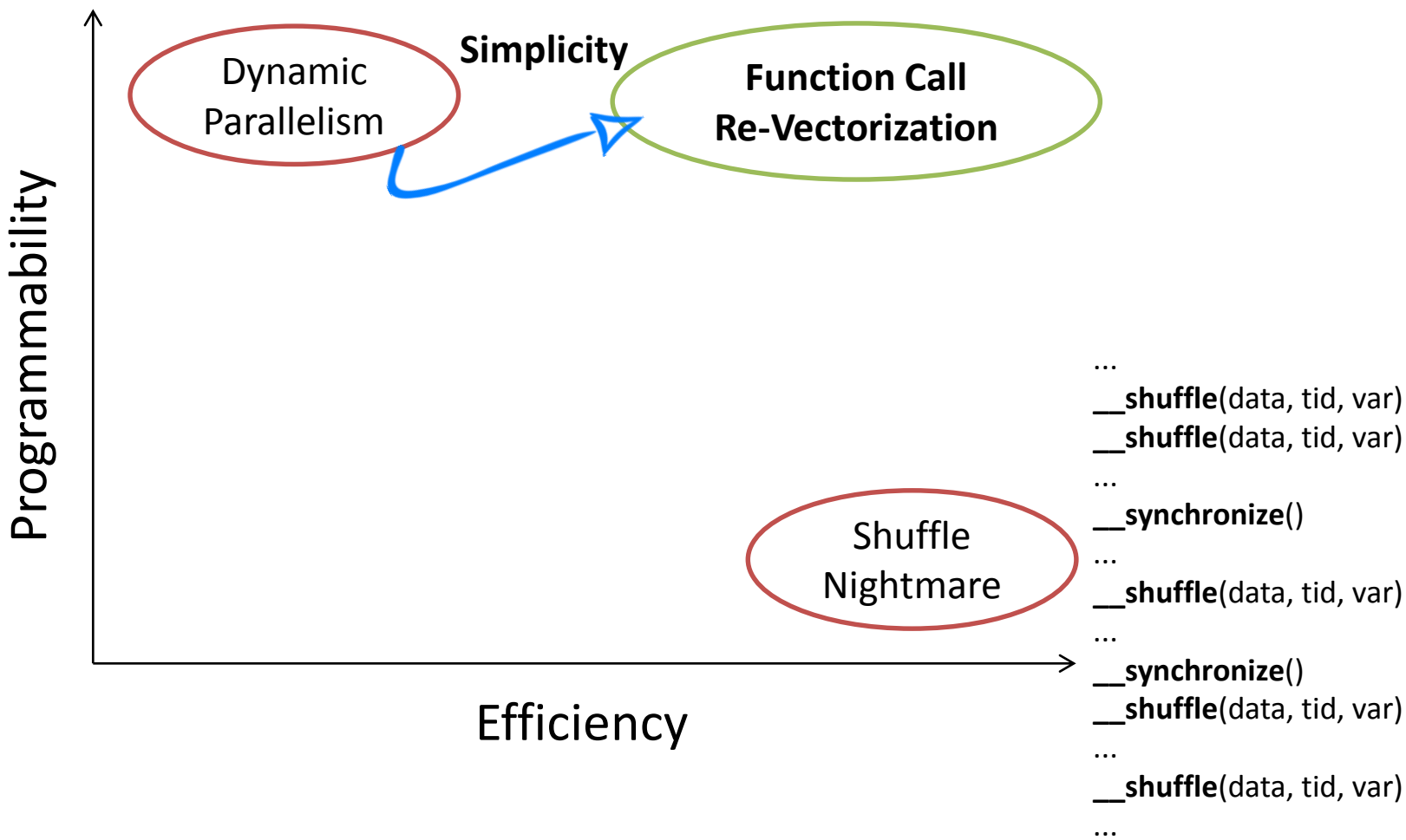
CUDA: kernel**<<<#warps, #threads>>>**(args...)



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

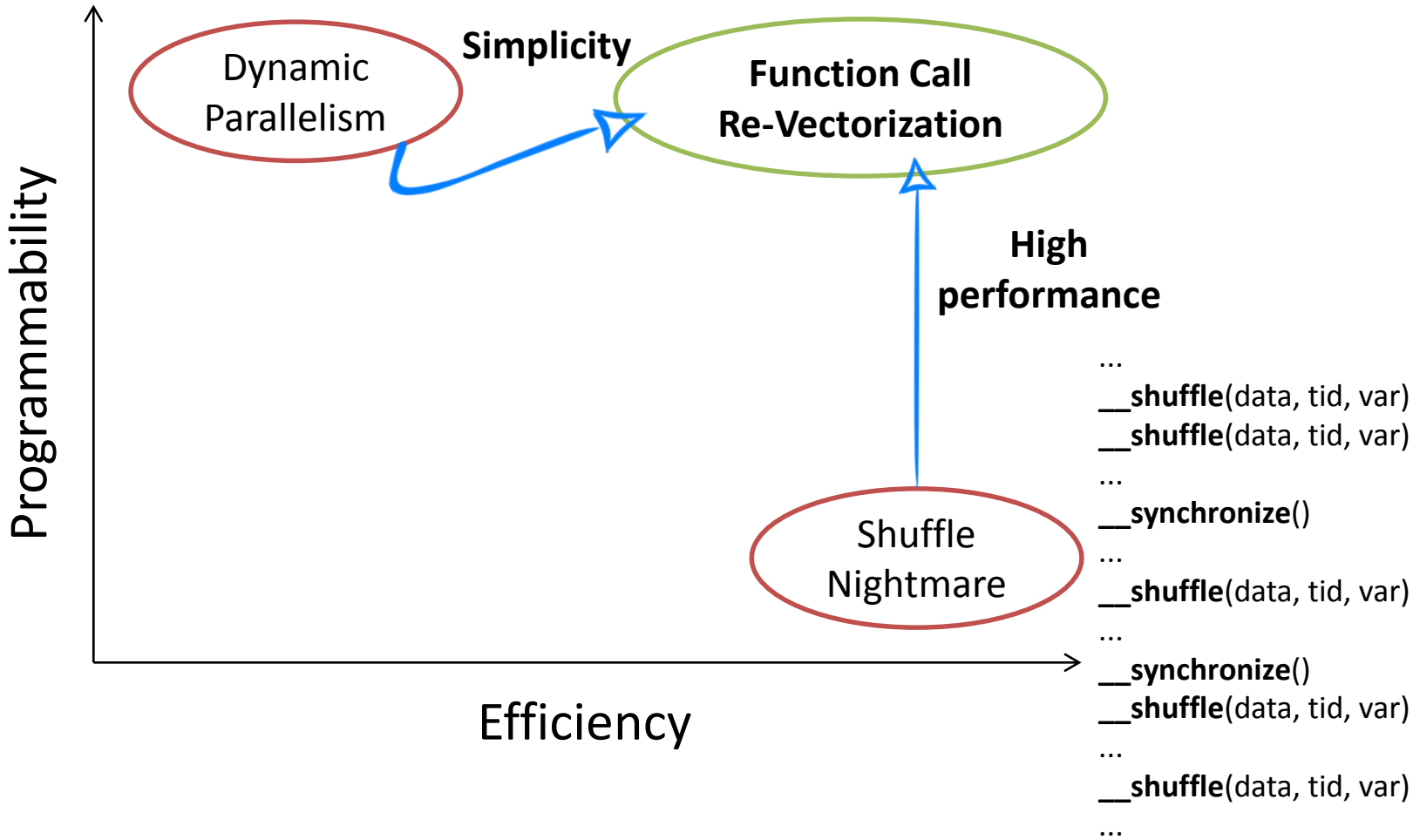
Function Call Re-Vectorization

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**

Too much **overhead**

```
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**

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]);
}
```

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);
    }
  }
}
```

Too many **lines of code**

Warp-synchronous wrapper for SIMD memory copy.

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

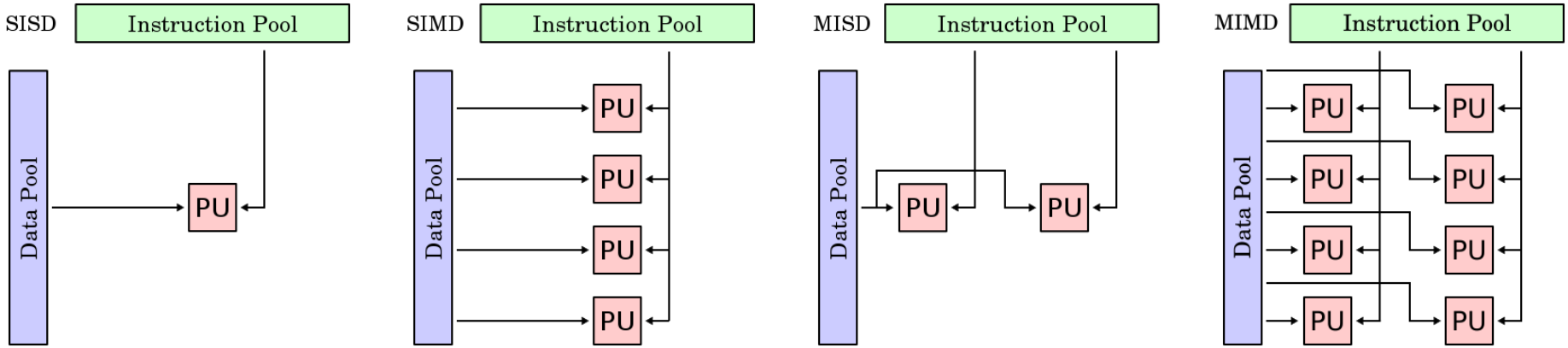
SIMD function calls
within *divergent regions*

Compilers Laboratory

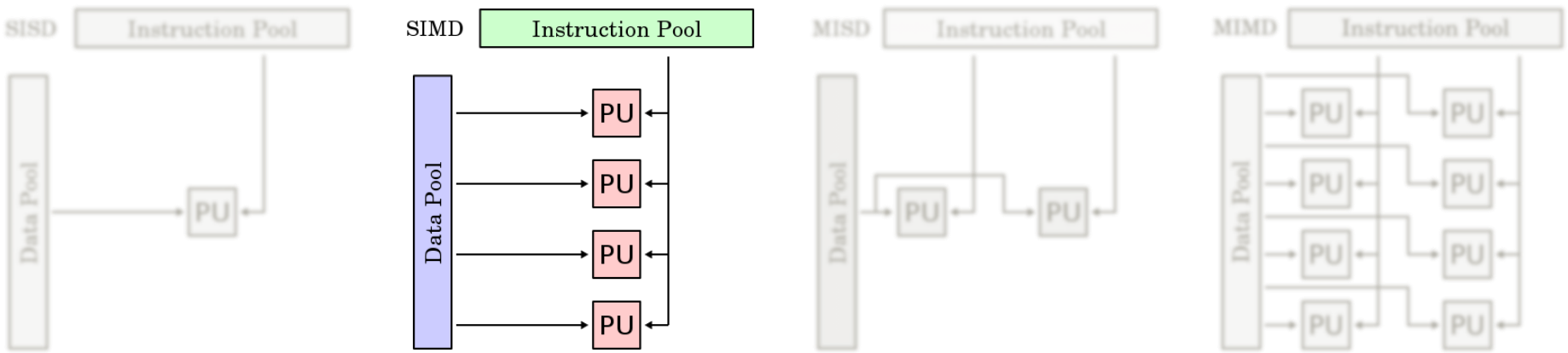
Department of Computer Science
Universidade Federal de Minas Gerais
Federal University of Minas Gerais, Brazil

Concepts

Concepts: Flynn's Taxonomy



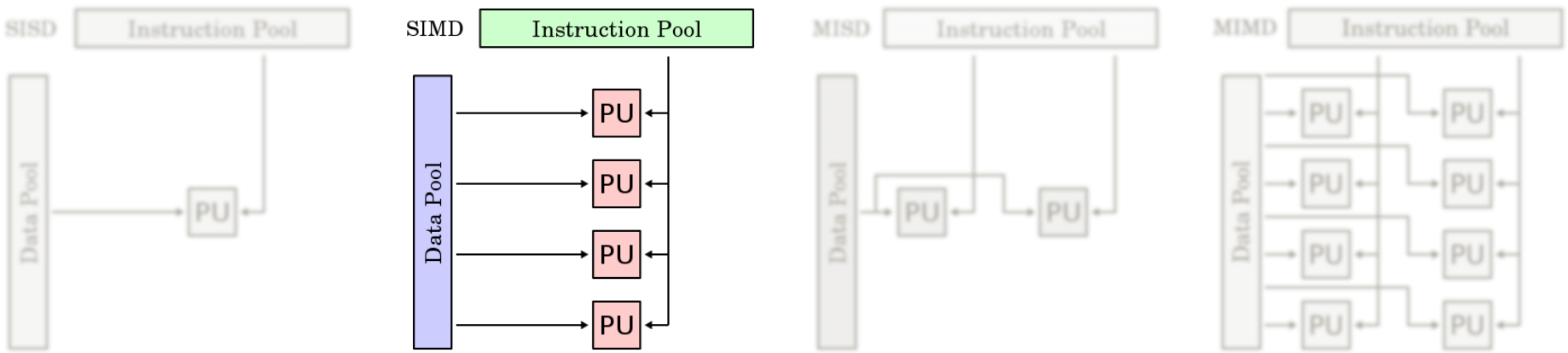
Concepts: Flynn's Taxonomy



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

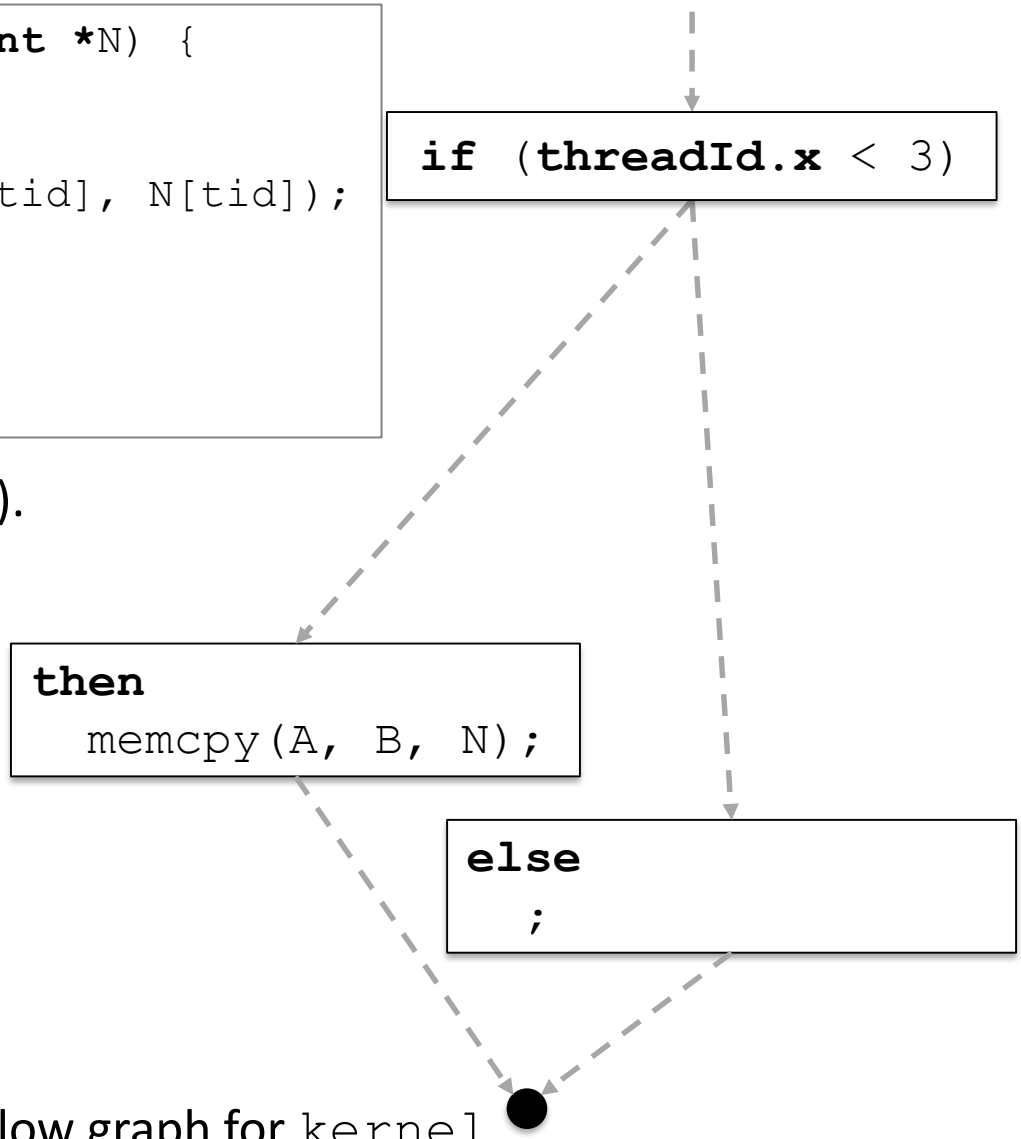
```
void kernel(int **A, int **B, int *N) {
  int tid(threadId.x);
  if (tid < 3) {
    memcpy<<<1, 4>>>(A[tid], B[tid], N[tid]);
  } else {
    ;
  }
}
```

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).

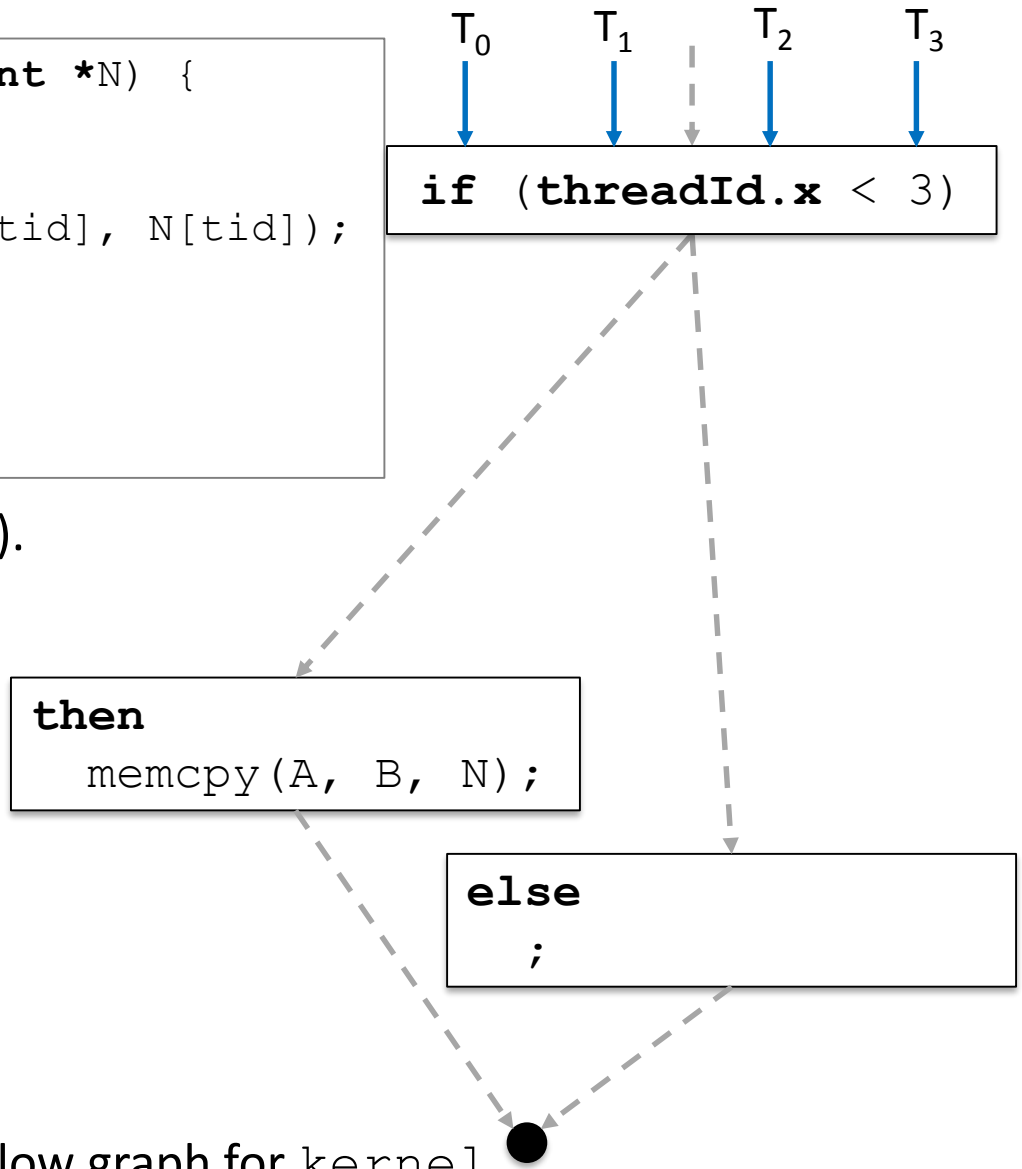


Control flow graph for `kernel`.

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).



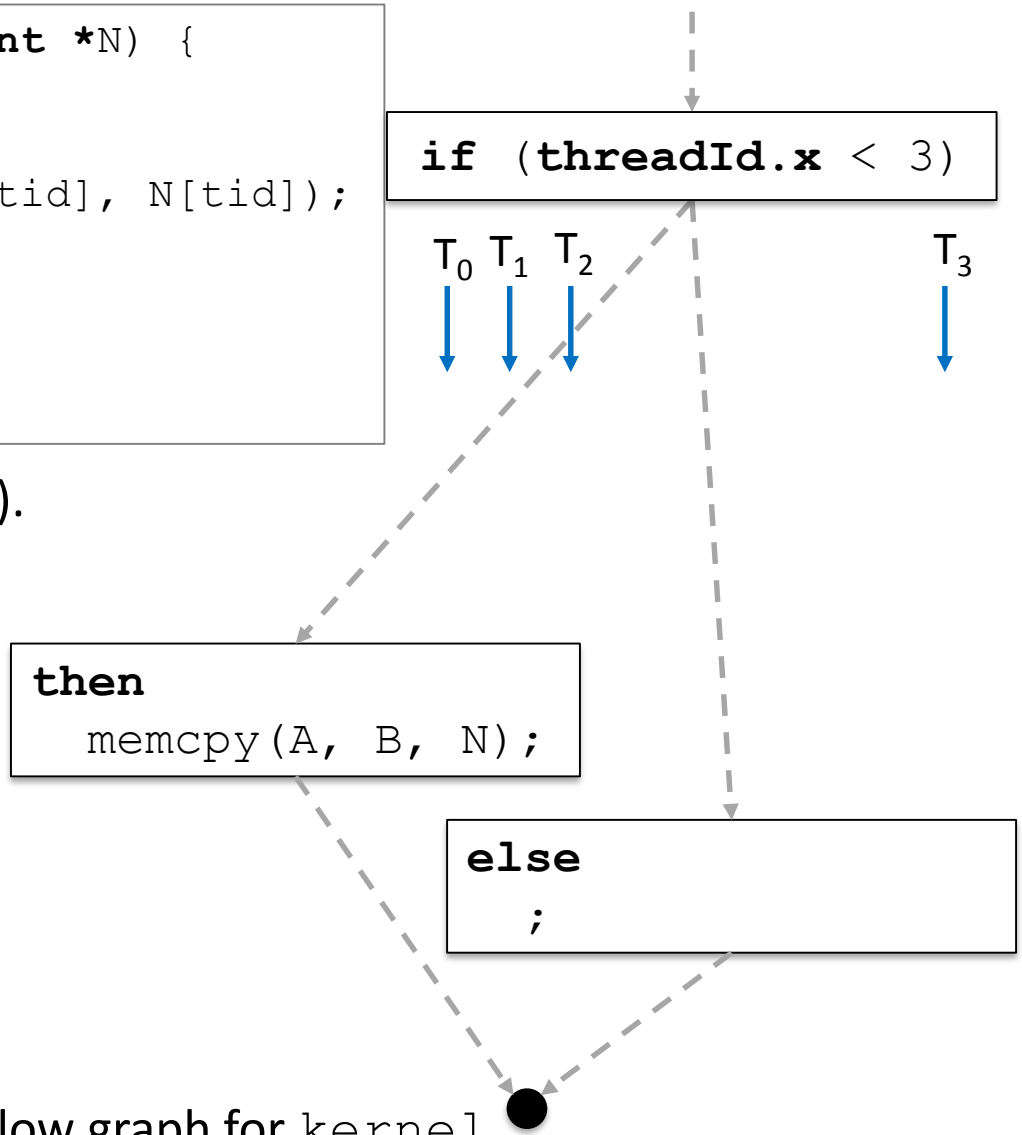
SIMD: LOCKSTEP EXECUTION!

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).



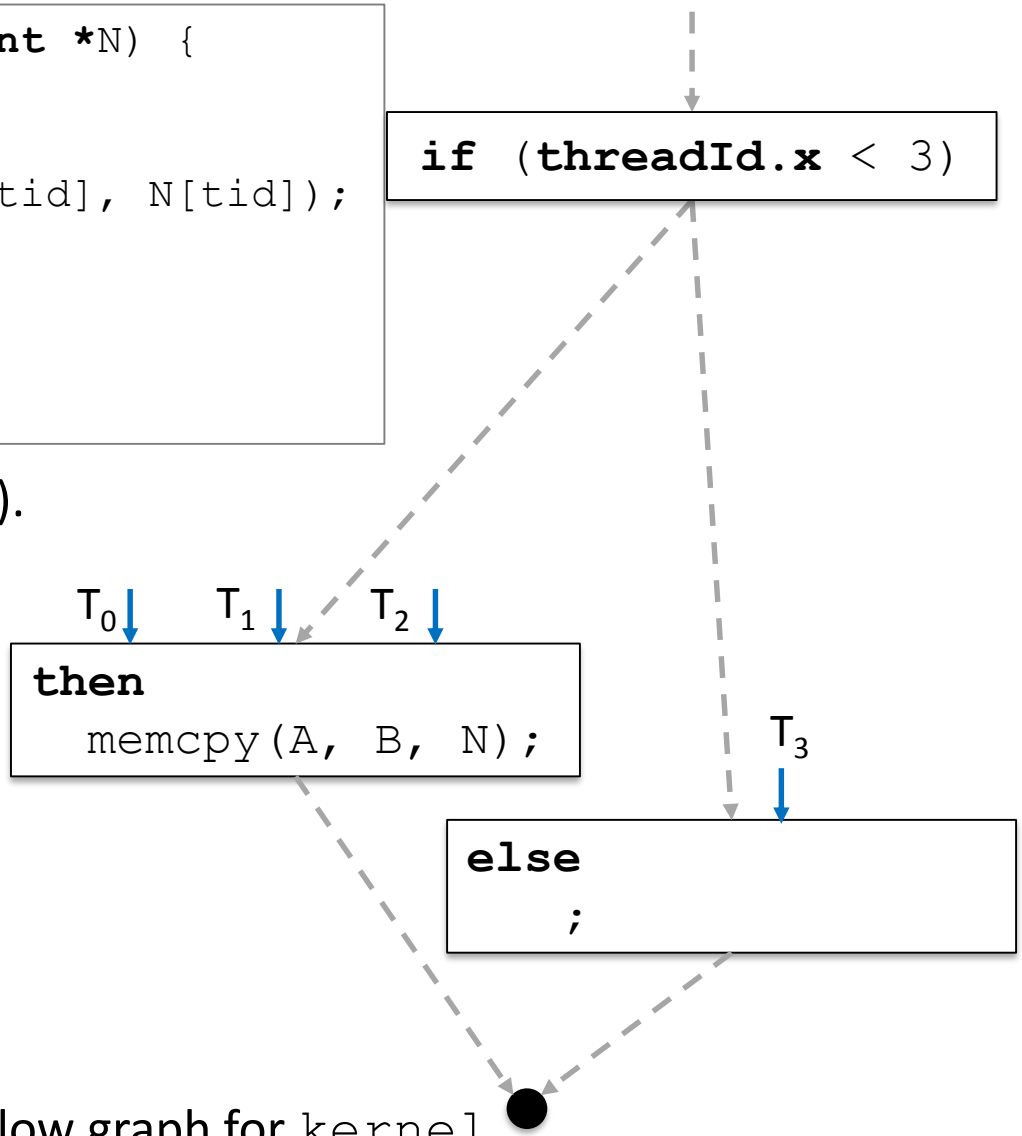
SIMD: LOCKSTEP EXECUTION!

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).



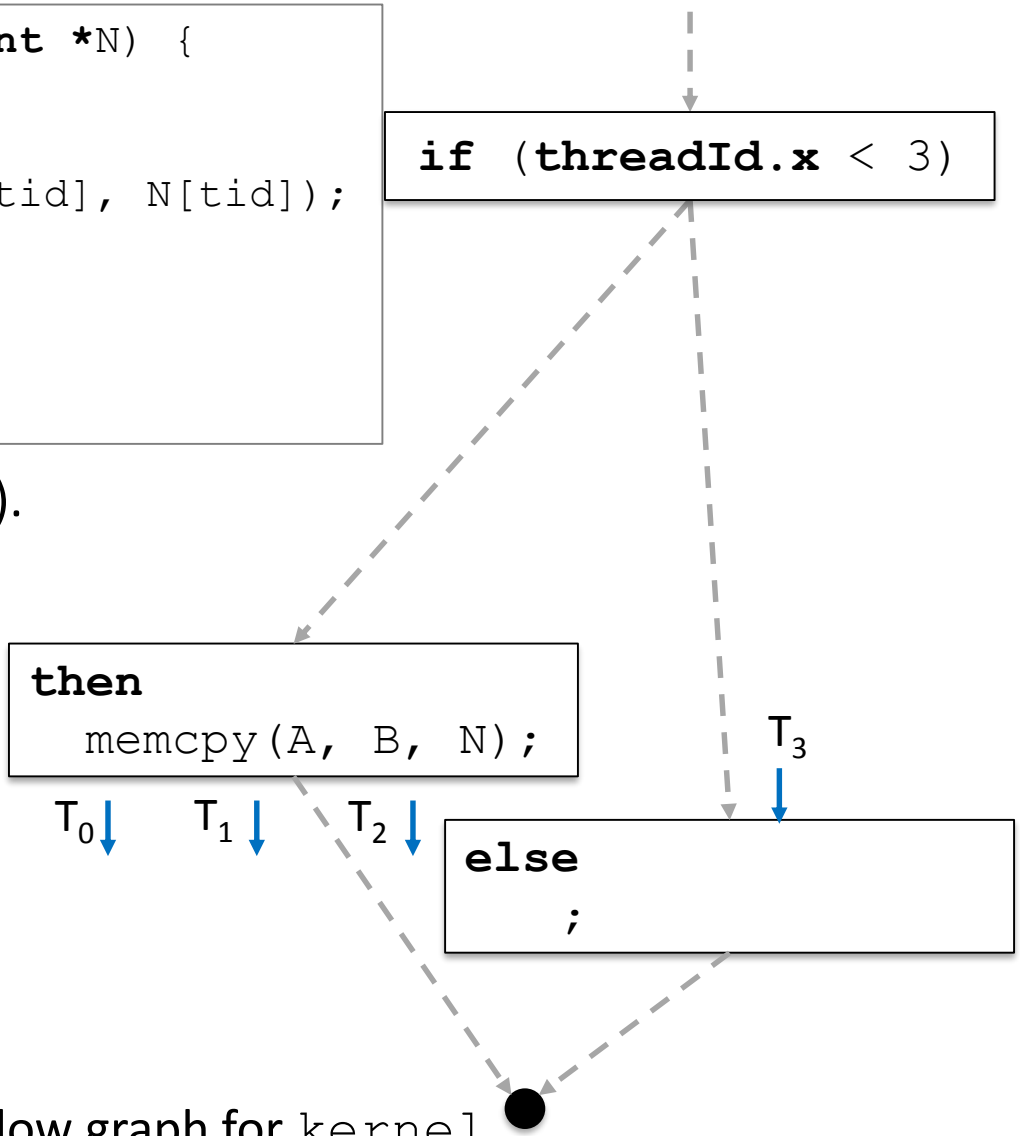
SIMD: LOCKSTEP EXECUTION!

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).



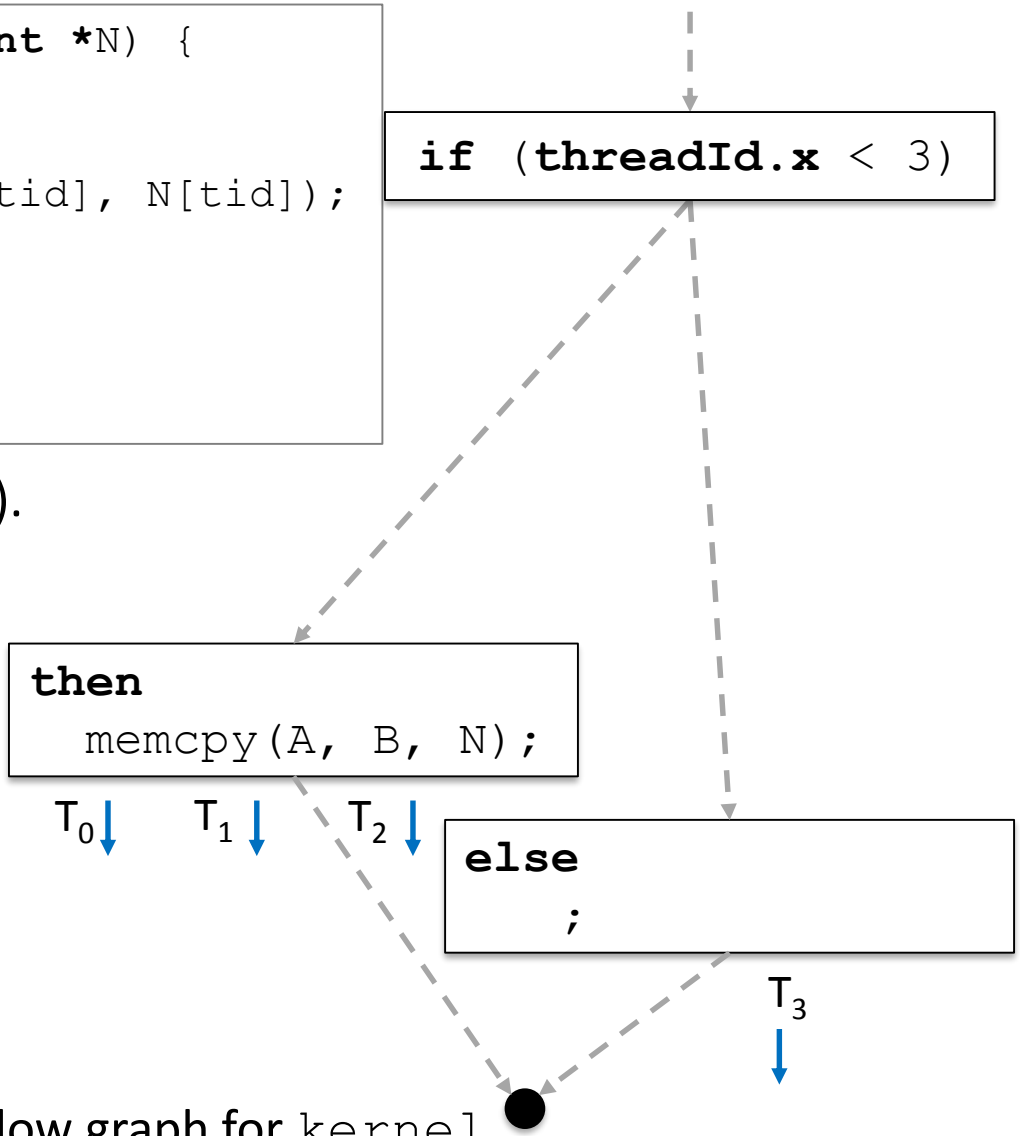
SIMD: LOCKSTEP EXECUTION!

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).



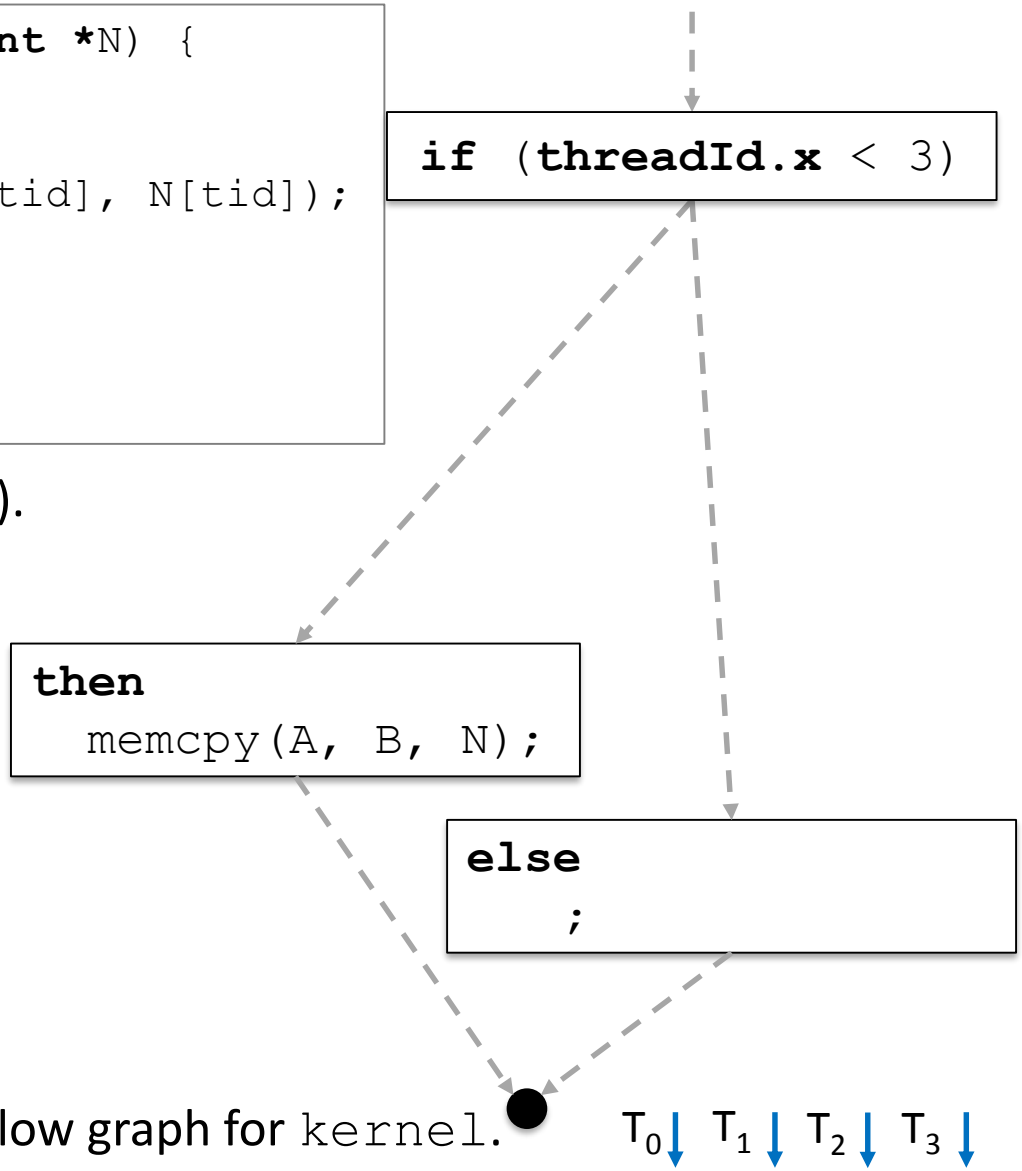
SIMD: LOCKSTEP EXECUTION!

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`.

Compilers Laboratory

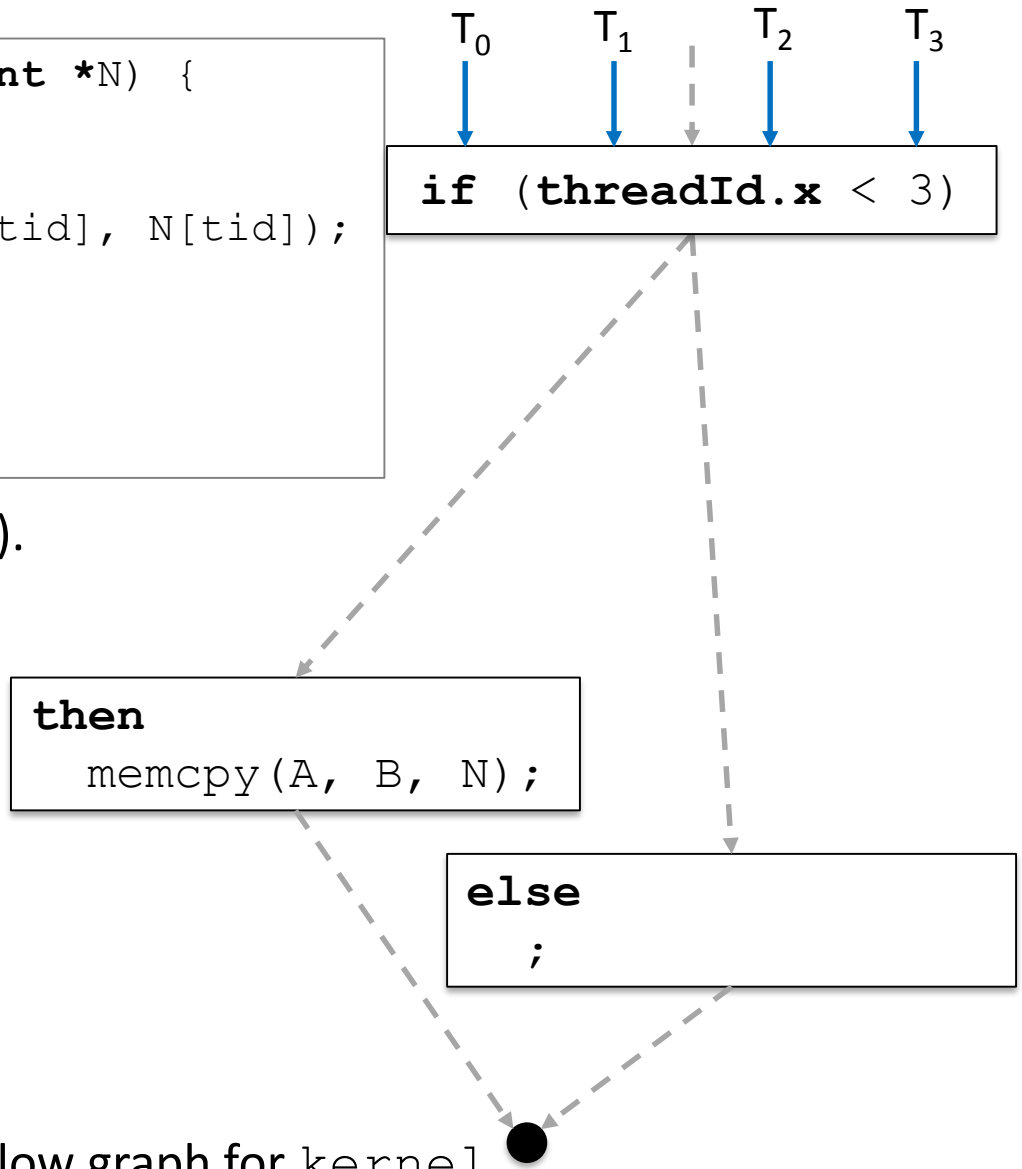
Department of Computer Science
Universidade Federal de Minas Gerais
Federal University of Minas Gerais, Brazil

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).



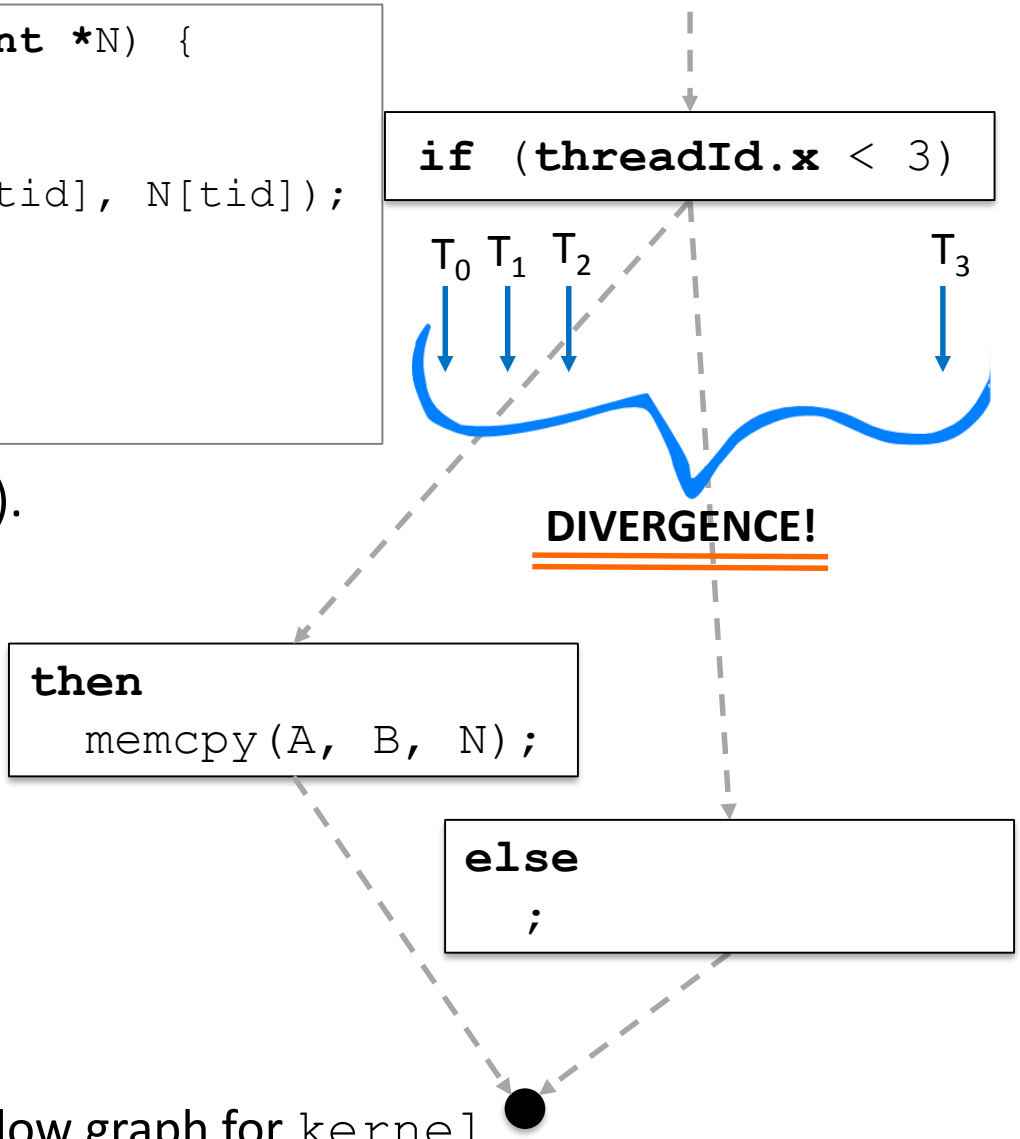
SIMD: LOCKSTEP EXECUTION!

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).



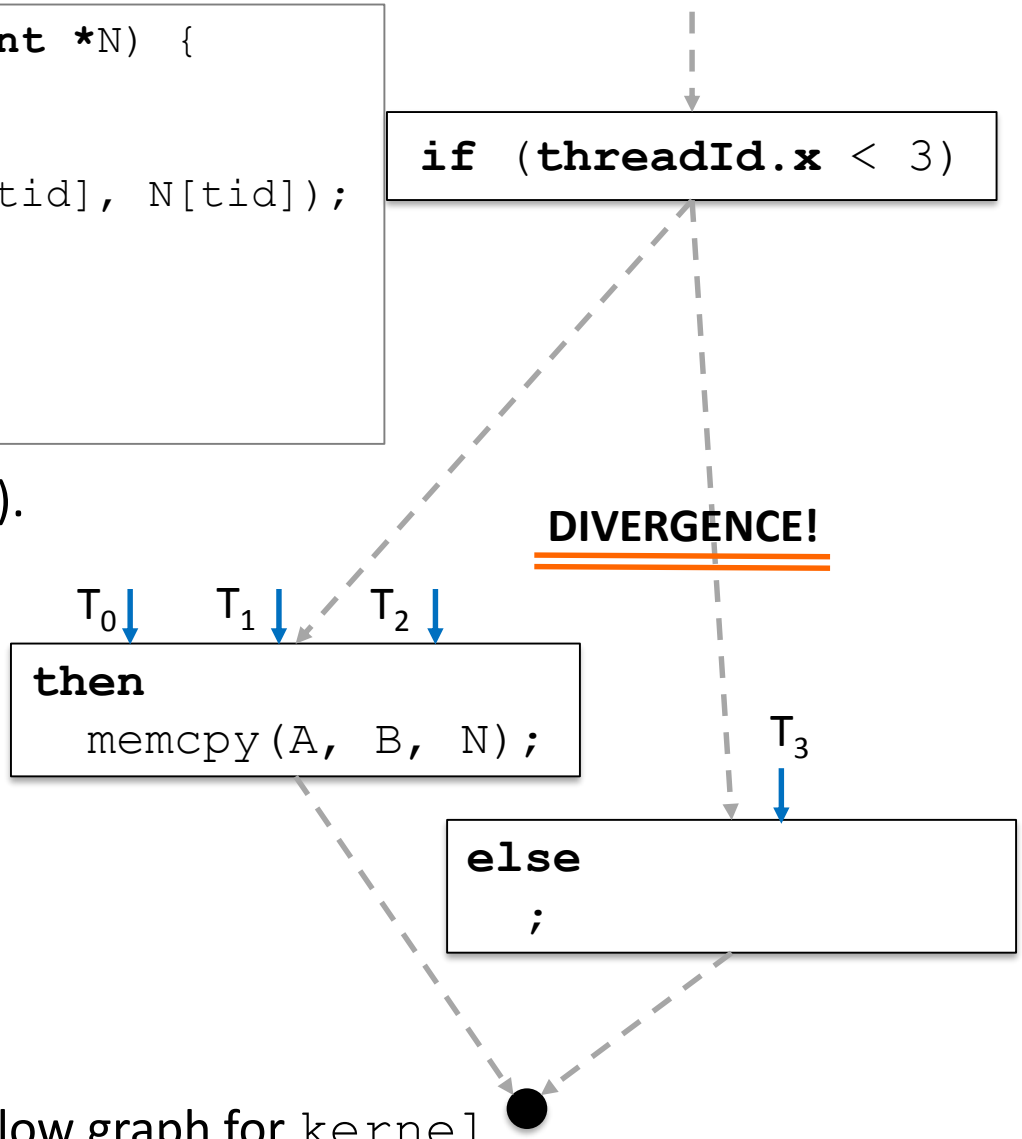
SIMD: LOCKSTEP EXECUTION!

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).



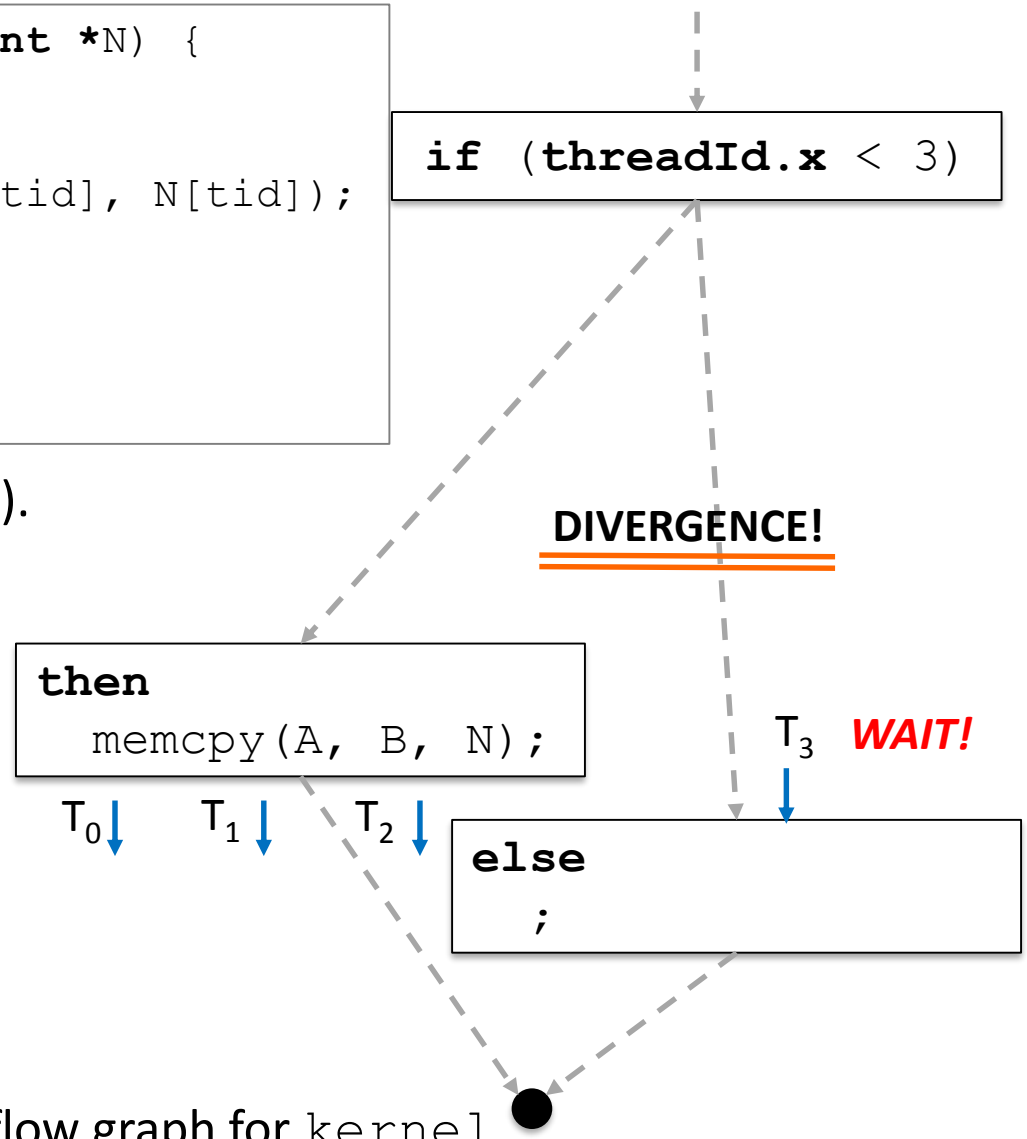
SIMD: LOCKSTEP EXECUTION!

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).



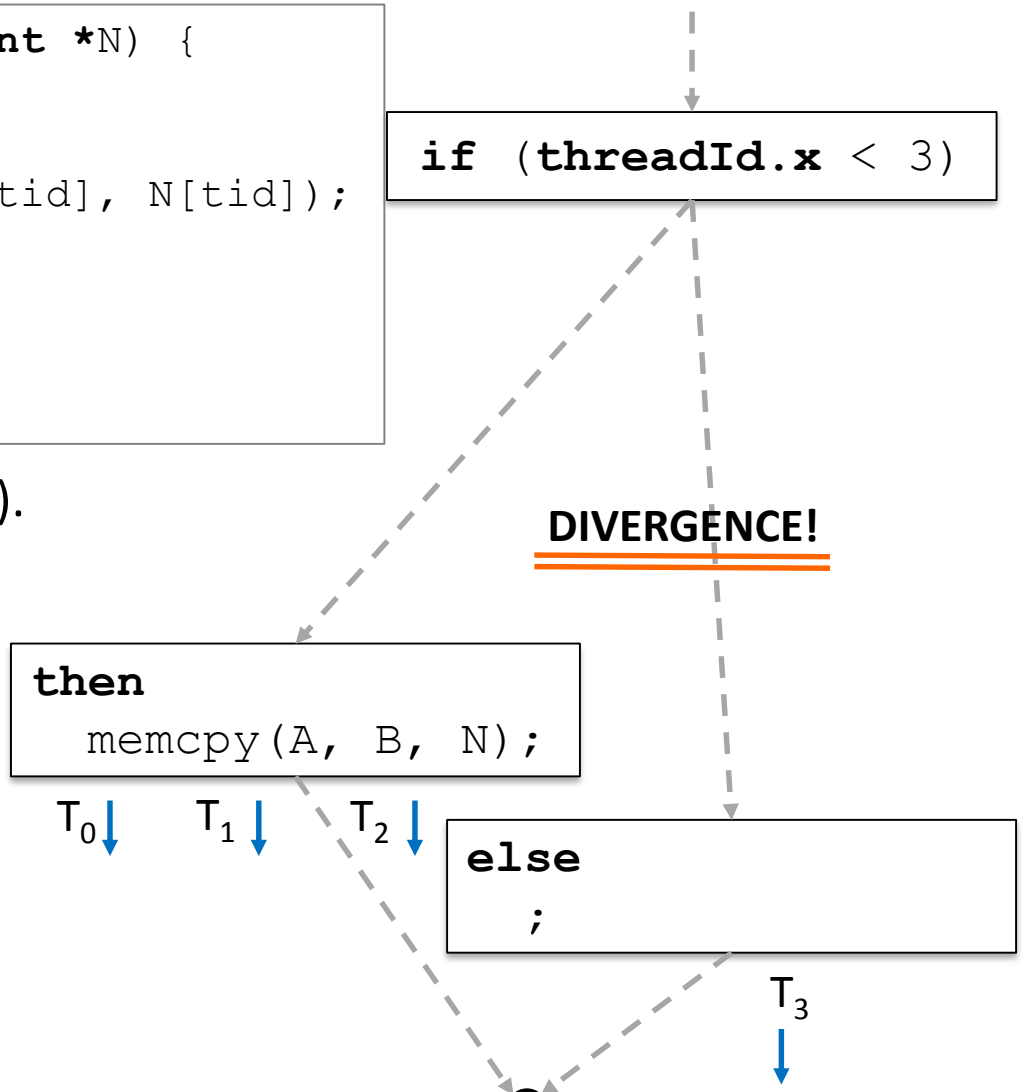
SIMD: LOCKSTEP EXECUTION!

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!

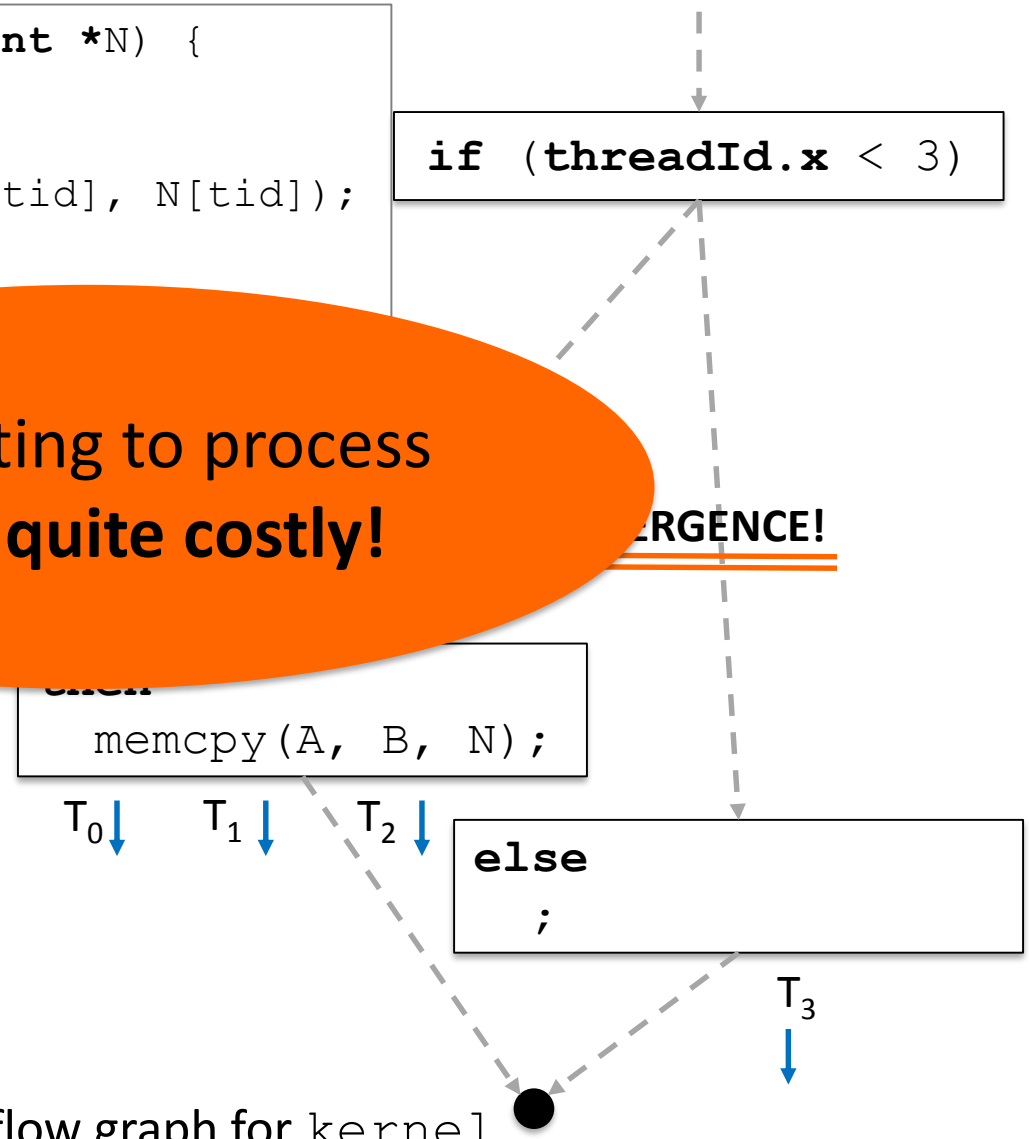
Control flow graph for `kernel`.

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 pa

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]--;
      }
   }
}
```

Source: http://homepages.dcc.ufmg.br/~fernando/classes/dcc888/ementa/slides/DivergenceAnalysis.pdf

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]--;
      }
   }
}
```

Source: http://homepages.dcc.ufmg.br/~fernando/classes/dcc888/ementa/slides/DivergenceAnalysis.pdf

Interlude: The Kernels of Samuel

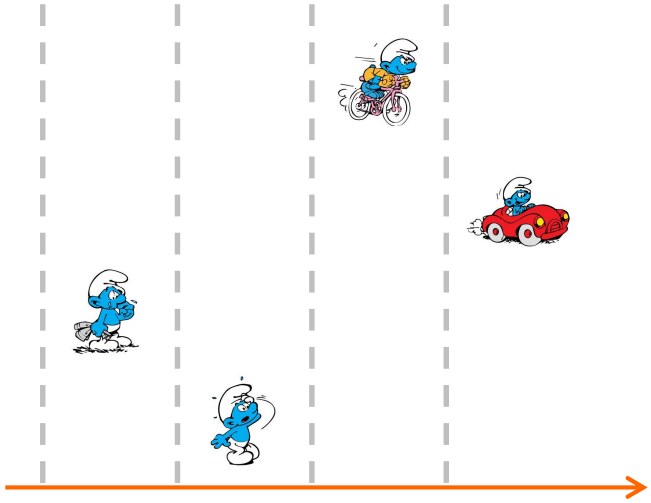
```
__global__ void dec2zero(int *data, int N) {
   int xIndex = blockIdx.x * blockDim.x + threadIdx.x;
   if (xIndex < N) {
      while (data[xIndex] > 0) {
         data[xIndex]--;
      }
   }
}
```

Source: http://homepages.dcc.ufmg.br/~fernando/classes/dcc888/ementa/slides/DivergenceAnalysis.pdf

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]--;
        }
    }
}
```



Source: http://homepages.dcc.ufmg.br/~fernando/classes/dcc888/ementa/slides/DivergenceAnalysis.pdf

Interlude: The Kernels of Samuel

```
int idx = threadId.x;
int dimx = threadDim.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]`**

Source: http://homepages.dcc.ufmg.br/~fernando/classes/dcc888/ementa/slides/DivergenceAnalysis.pdf

Interlude: The Kernels of Samuel

```
int idx = threadId.x;
int dimx = threadDim.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]`**

Source: http://homepages.dcc.ufmg.br/~fernando/classes/dcc888/ementa/slides/DivergenceAnalysis.pdf

Interlude: The Kernels of Samuel

```
int idx = threadId.x;
int dimx = threadDim.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**

Source: http://homepages.dcc.ufmg.br/~fernando/classes/dcc888/ementa/slides/DivergenceAnalysis.pdf

Interlude: The Kernels of Samuel

```
int idx = threadId.x;
int dimx = threadDim.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]**

Source: http://homepages.dcc.ufmg.br/~fernando/classes/dcc888/ementa/slides/DivergenceAnalysis.pdf

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]--;
        }
    }
}
```

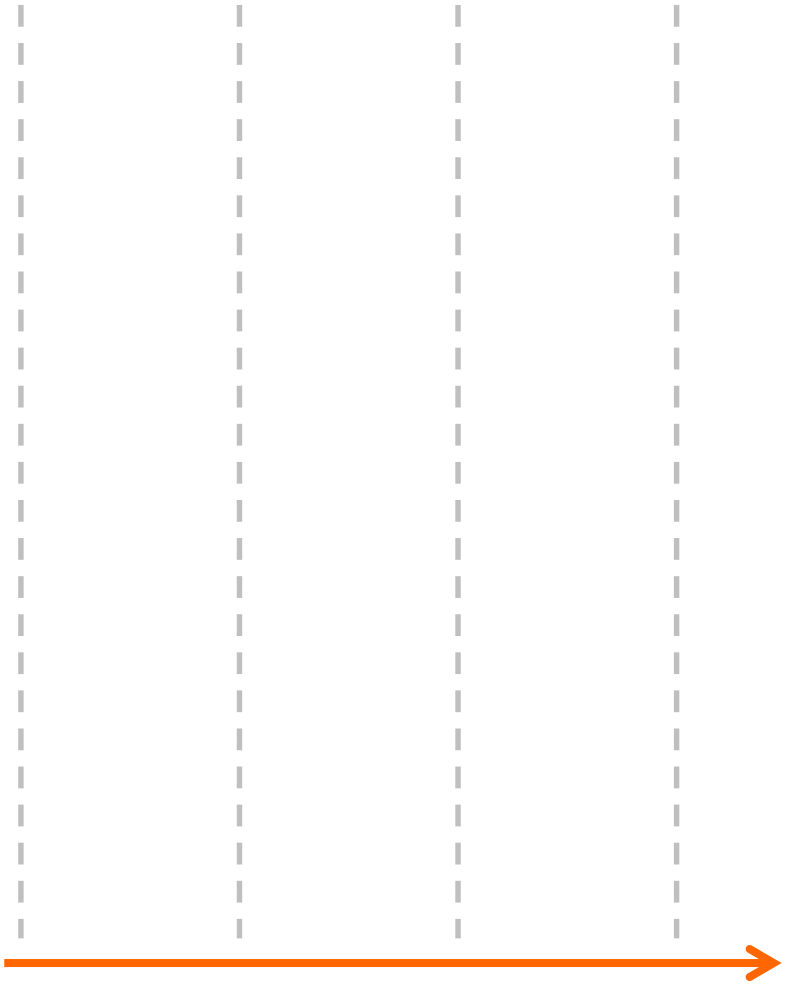
```
int idx = threadId.x;
int dimx = threadDim.x;
```

```
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;
  }
}
```



Source: http://homepages.dcc.ufmg.br/~fernando/classes/dcc888/ementa/slides/DivergenceAnalysis.pdf

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]--;
        }
    }
}
```

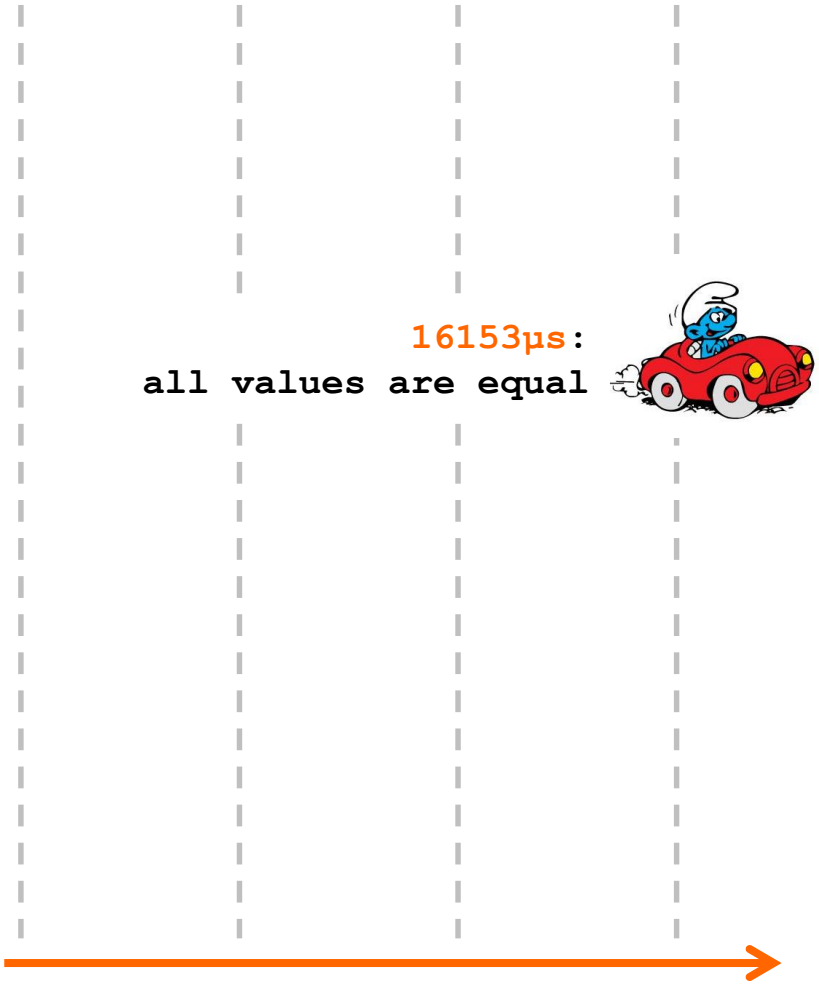
```
int idx = threadId.x;
int dimx = threadDim.x;
```

```
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;
  }
}
```



Source: http://homepages.dcc.ufmg.br/~fernando/classes/dcc888/ementa/slides/DivergenceAnalysis.pdf

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]--;
        }
    }
}
```

```
int idx = threadId.x;
int dimx = threadDim.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;
  }
}
```

16250µs:
values differ by constant

16153µs:
all values are equal

Source: http://homepages.dcc.ufmg.br/~fernando/classes/dcc888/ementa/slides/DivergenceAnalysis.pdf

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]--;
        }
    }
}
```

```
int idx = threadId.x;
int dimx = threadDim.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;
  }
}
```

16250µs:
values differ by constant

16153µs:
all values are equal

30210µs:
normal distribution of values

Source: http://homepages.dcc.ufmg.br/~fernando/classes/dcc888/ementa/slides/DivergenceAnalysis.pdf

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]--;
        }
    }
}
```

```
int idx = threadId.x;
int dimx = threadDim.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;
  }
}
```

16250µs:
values differ by constant

16153µs:
all values are equal

32193µs:
half the values differ!

30210µs:
normal distribution of values

Source: http://homepages.dcc.ufmg.br/~fernando/classes/dcc888/ementa/slides/DivergenceAnalysis.pdf

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]--;
        }
    }
}
```

Divergence is harmful to performance!

```
int idx
int

                                    {
  }
}
```

```
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;
  }
}
```

16250µs: values differ by constant

16153µs: all values are equal

32193µs: half the values differ!

30210µs: normal distribution of values

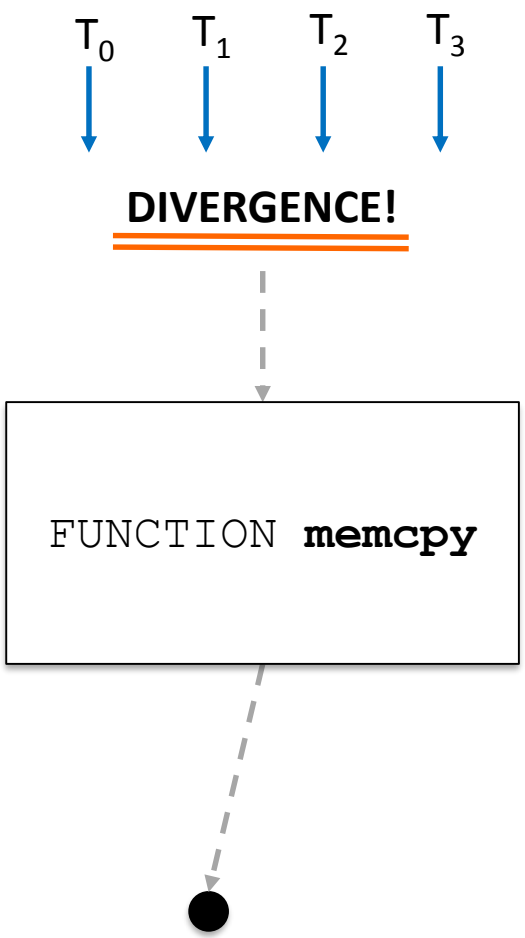
Source: http://homepages.dcc.ufmg.br/~fernando/classes/dcc888/ementa/slides/DivergenceAnalysis.pdf

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**



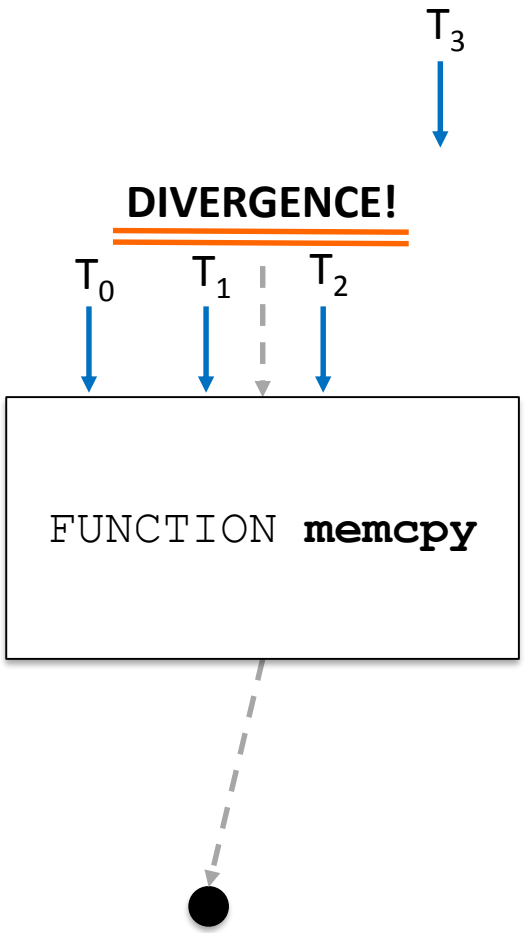
Control flow graph for `memcpy`.

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**



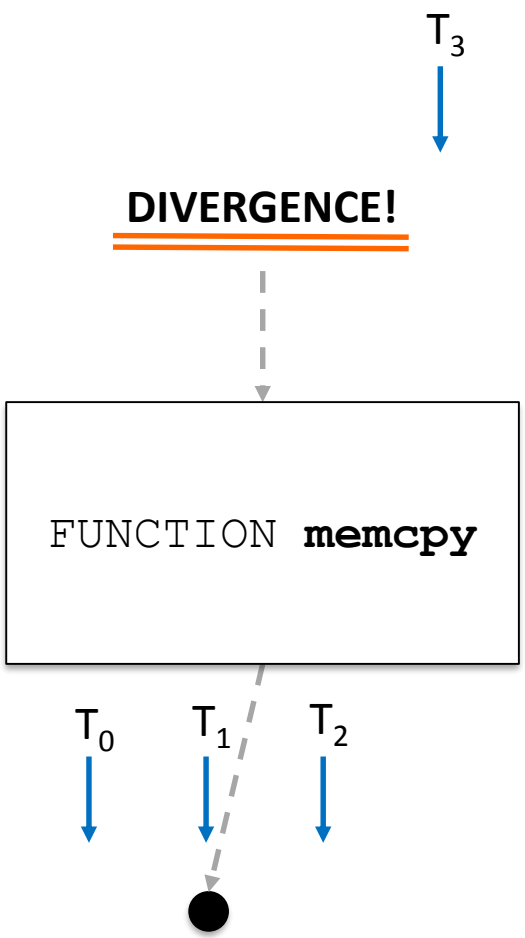
Control flow graph for `memcpy`.

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`.

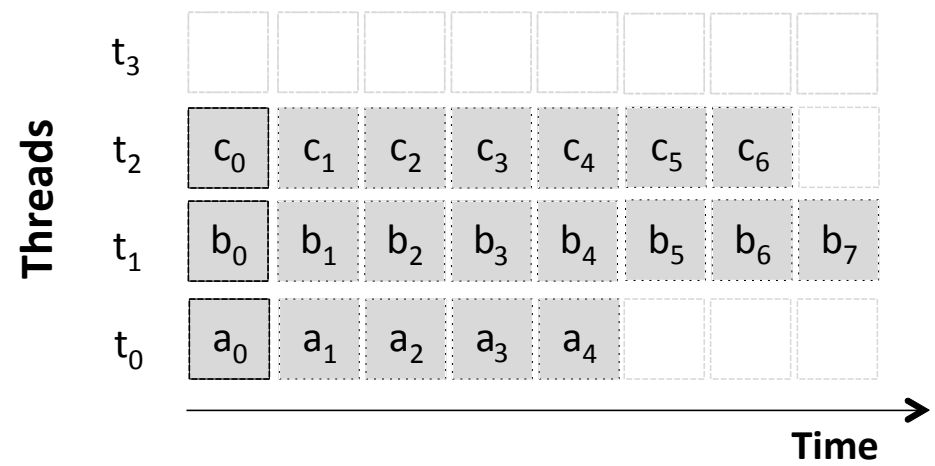
Divergences: *Coda*

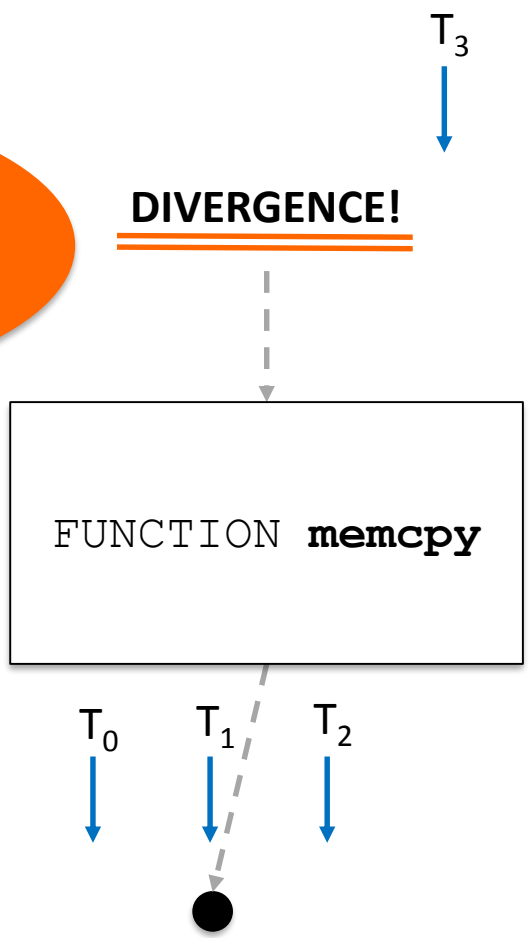
```
void kernel(int **A, int **B, int *N) {
  int tid(threadId.x);
  if (tid < 3) {
    memcpy<<<1, 4>>>(A[tid], B[tid], N[tid]);
  } else {
    ;
  }
}
```

Kernel for parallel executi

Suboptimal behavior:
thread **T_3 is inactive**.
Right?

Observed behavior:





Control flow graph for `memcpy`.

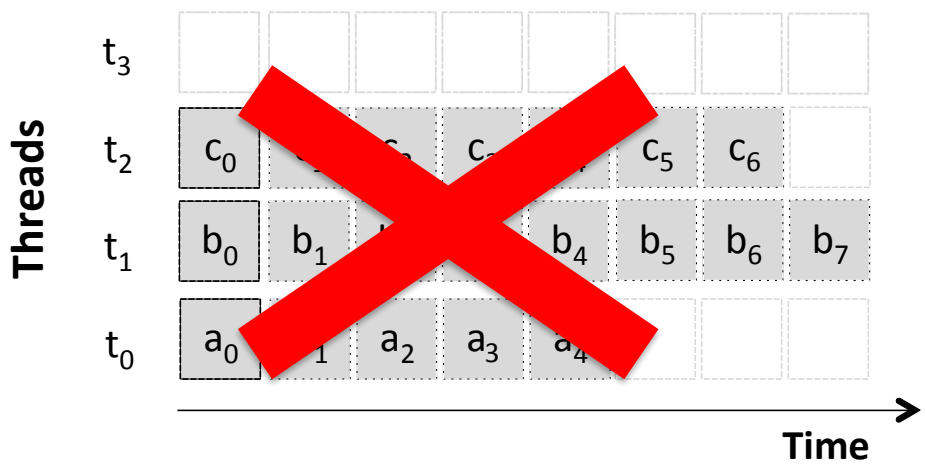
Divergences: *Coda*

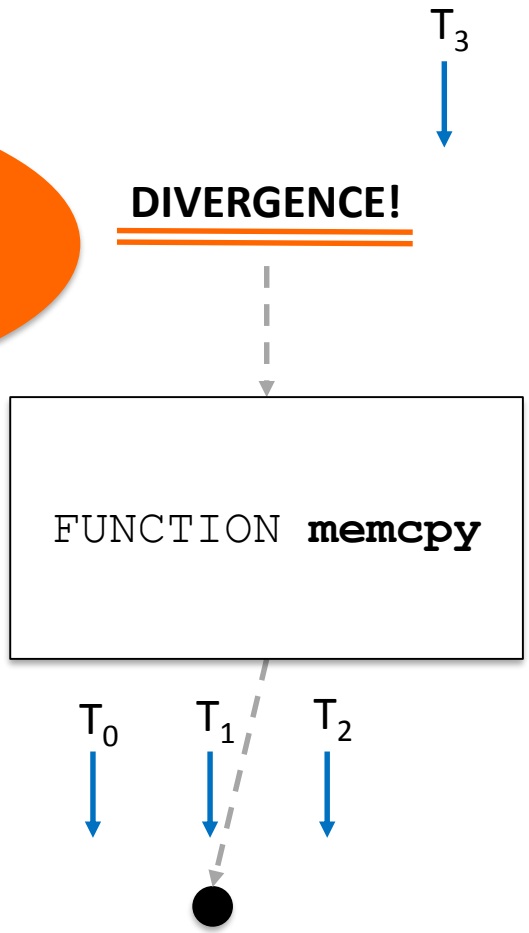
```
void kernel(int **A, int **B, int *N) {
  int tid(threadId.x);
  if (tid < 3) {
    memcpy<<<1, 4>>>(A[tid], B[tid], N[tid]);
  } else {
    ;
  }
}
```

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

Kernel for parallel execuli

Observed behavior:





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

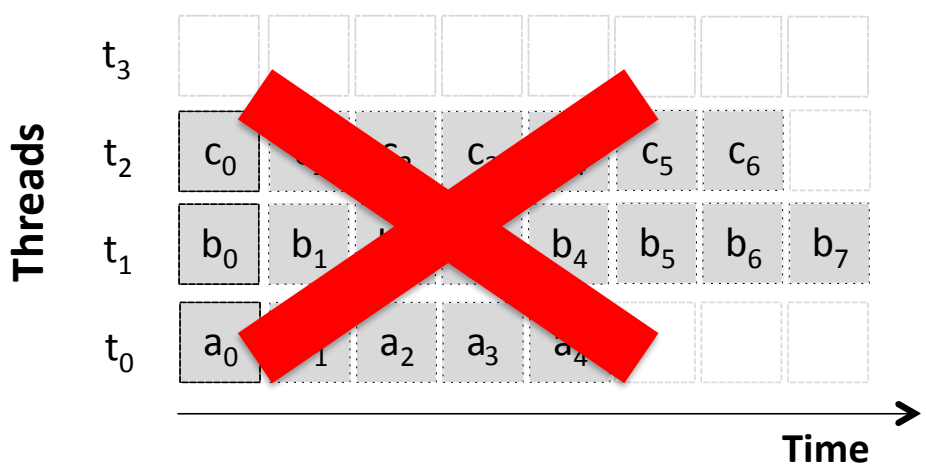
Dynamic Parallelism

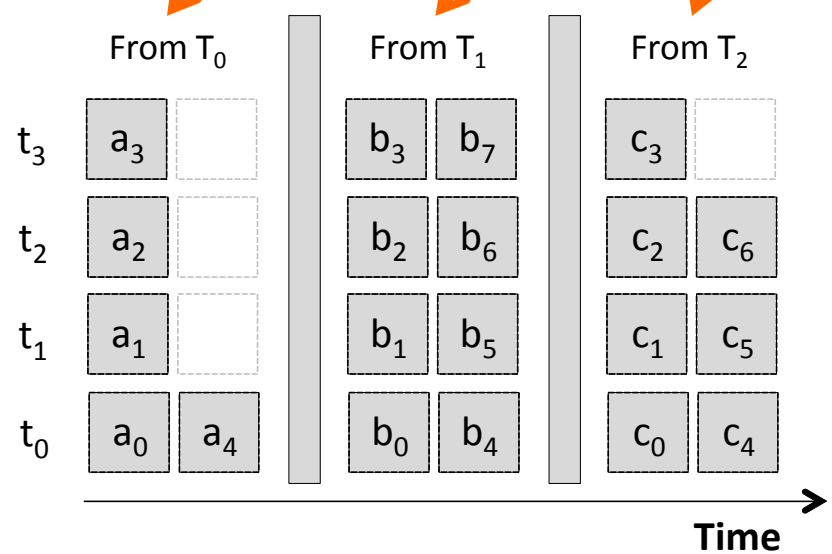
```
void kernel(int **A, int **B, int *N) {
  int tid(threadId.x);
  if (tid < 3) {
    memcpy<<<1, 4>>>(A[tid], B[tid], N[tid]);
  } else {
    ;
  }
}
```

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

Kernel for parallel execution (CUDA).

Actual behavior with CUDA's dynamic parallelism:





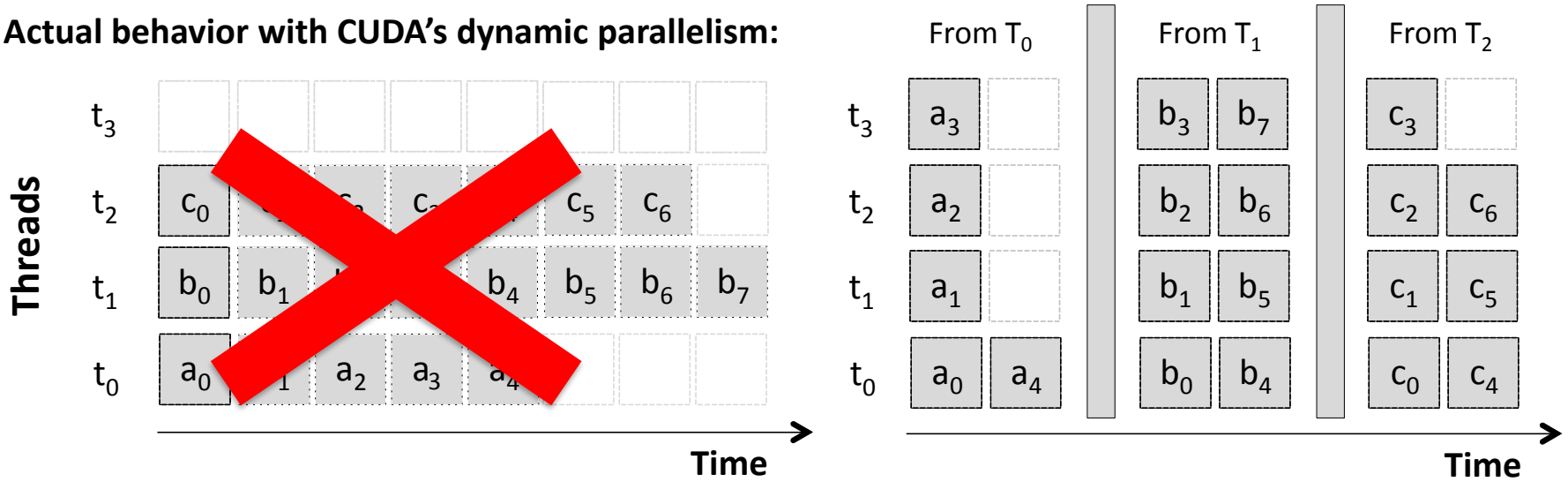
Dynamic Parallelism

```
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:

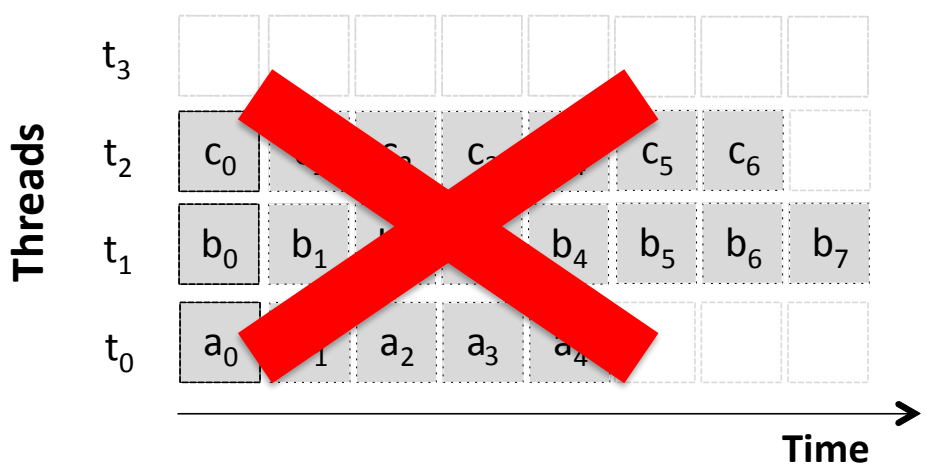


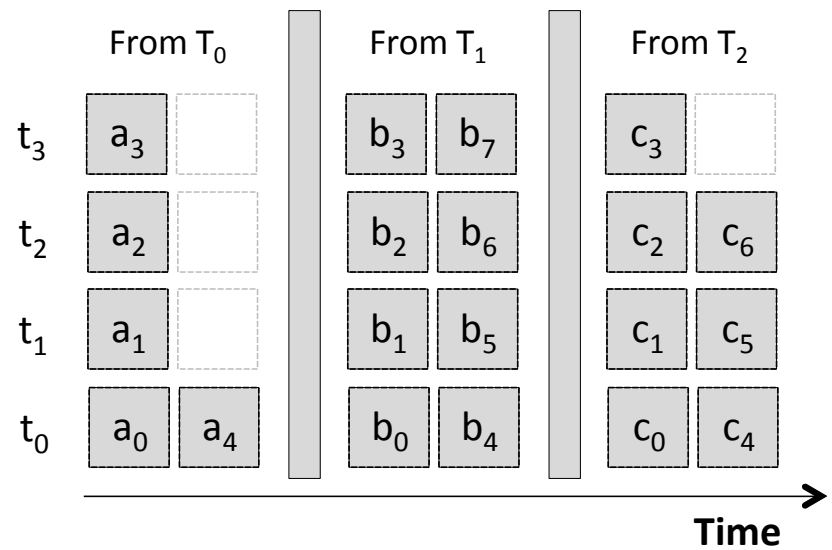
Dynamic Parallelism

```
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:





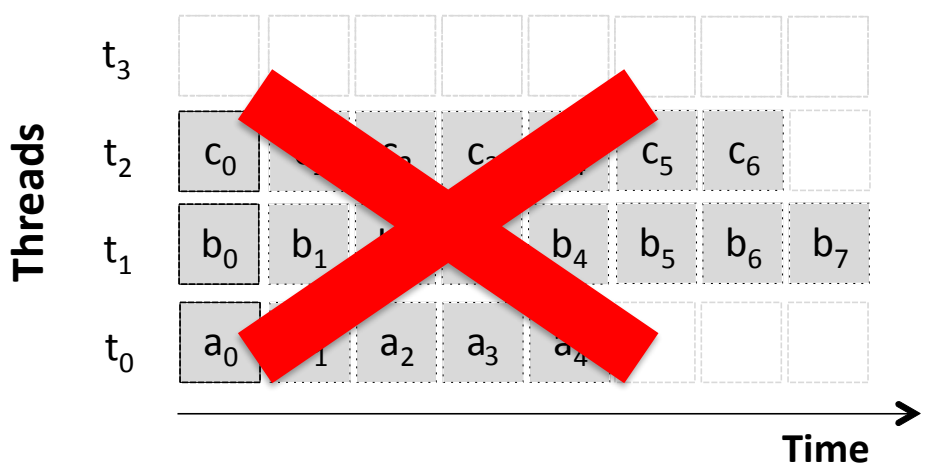
Dynamic Parallelism

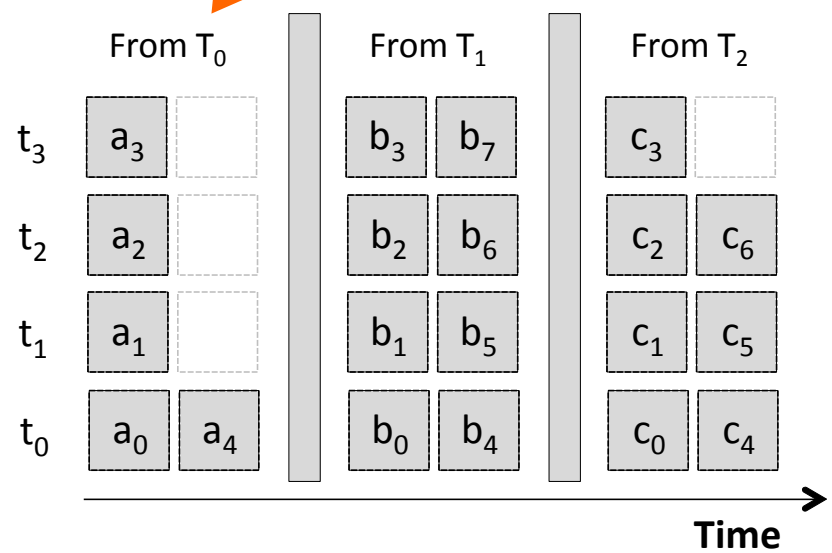
```
void memcpy(int *dest, int *src, int N) {
  for (int i=threadId.x; i < N; i+=threadDim.x) {
    dest[i] = src[i];
  }
}
```

SIMD implementation of memory copy.

All threads work on a single vector!

Actual behavior with CUDA's dynamic parallelism:





Dynamic Parallelism

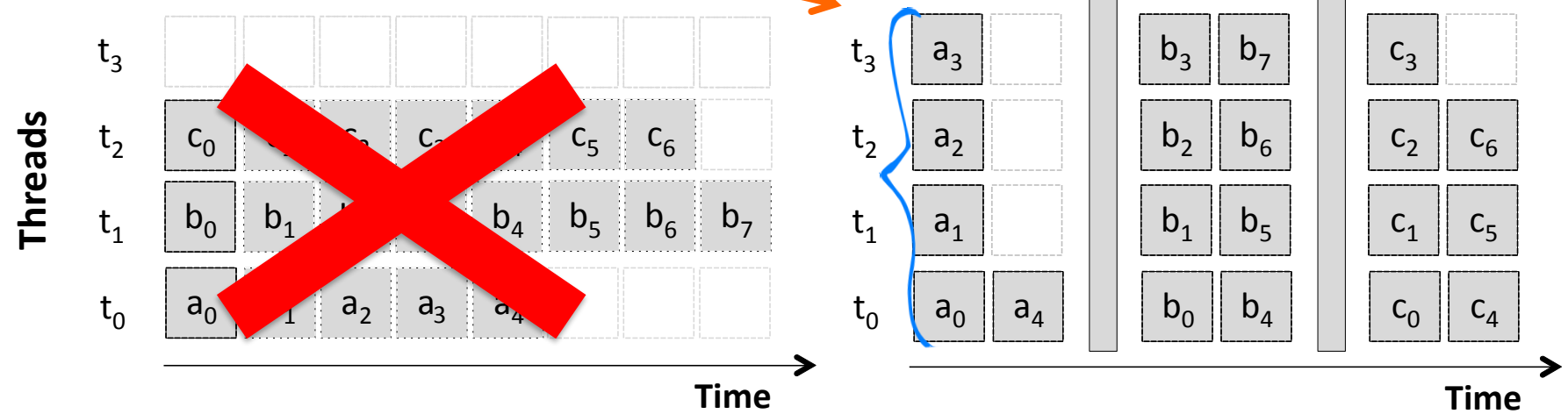
```
void memcpy(int *dest, int *src, int N) {
  for (int i=threadId.x; i < N; i+=threadDim.x) {
    dest[i] = src[i];
  }
}
```

All threads work on a single vector!

SIMD implementation of memory copy.

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

Actual behavior with CUDA's dynamic parallelism:



Dynamic Parallelism

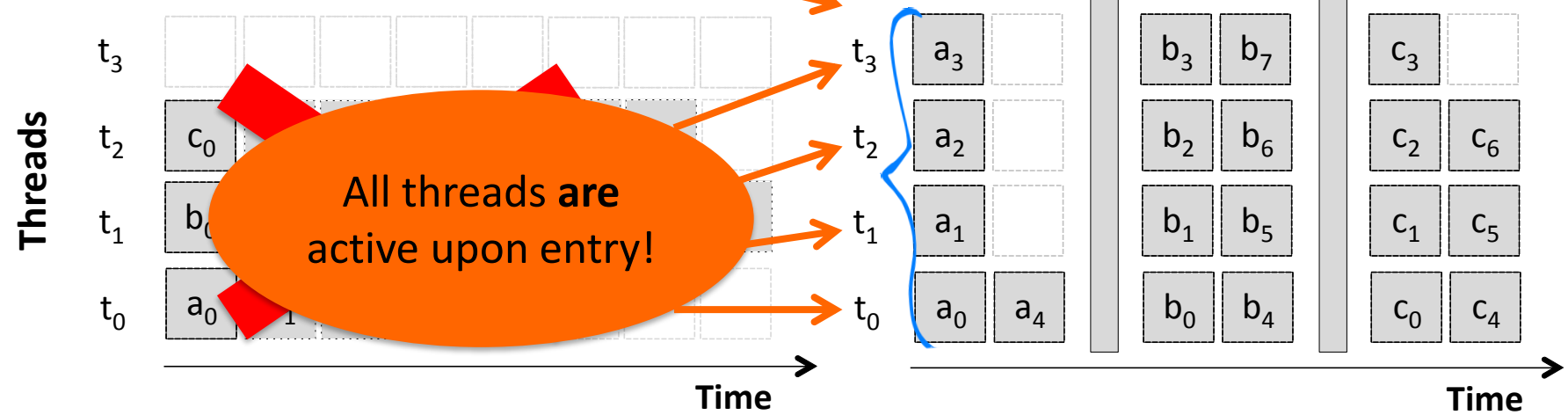
```
void memcpy(int *dest, int *src, int N) {
  for (int i=threadId.x; i < N; i+=threadDim.x) {
    dest[i] = src[i];
  }
}
```

SIMD implementation of memory copy.

All threads work on a single vector!

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

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 {
    ;
  }
}
```

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

Kernel for parallel execution (CUDA).

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

```
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 {
    ;
  }
}
```

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

Kernel for parallel execution (CUDA).

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

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

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

Dynamic Parallelism

```
void
  int
  if
  }

  }
}
```

Ker

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/CUDADownloads/TechBrief_Dynamic_Parallelism_in_CUDA.pdf

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

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

Compilers Laboratory

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:

T_0 T_1 T_2 T_3

int value = [10 20 30 10] → increment → **int** value = [11 21 31 11]

T_0 T_1 T_2 T_3

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_0 T_1 T_2 T_3

int value = [10 20 30 10] → increment → T_0 T_1 T_2 T_3 **int** value = [11 21 31 11]

Reductions:

T_0 T_1 T_2 T_3

int value = [10 20 30 10] → sum → T_0 T_1 T_2 T_3 **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**:

DIVERGENCE: T_0 T_1 T_2 T_3

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_0 T_1 T_2 T_3

EVERYWHERE: T_0 T_1 T_2 T_3

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

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_0 T_1 T_2 T_3

EVERYWHERE: T_0 T_1 T_2 T_3

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

DIVERGENCE: T_0 T_1 T_2 T_3

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

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 me

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 me

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

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

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 (i
      i
      mem
    }
  }
}
```

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

SPMD/SIMT

handle divergences

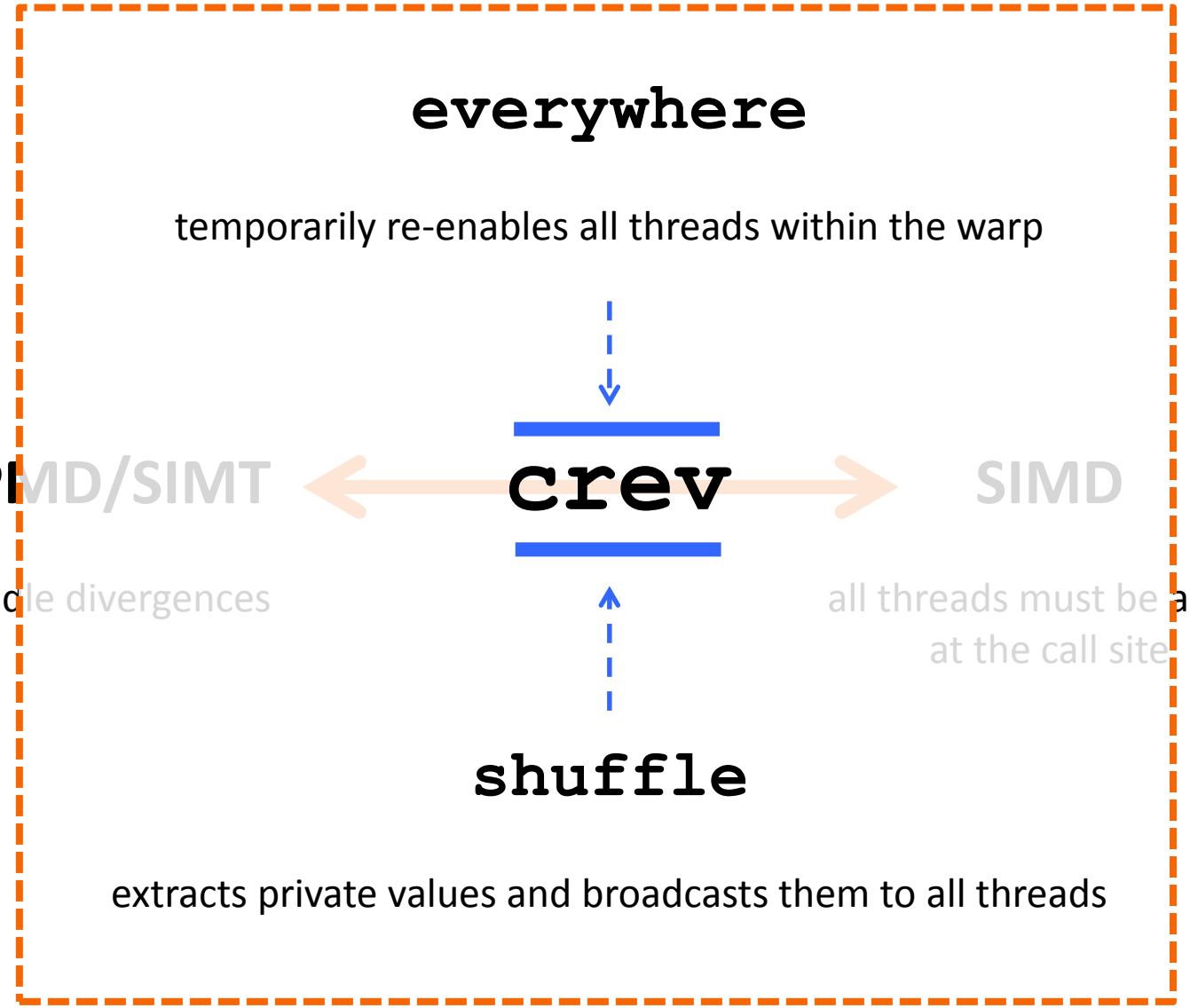
SIMD

all threads must be active
at the call site

`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:

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

$$(\textsc{Bt})\quad \frac{\begin{array}{c} P[pc] = \texttt{bz}\ v, l \qquad \textbf{split}(\Theta, \beta, v) = (\Theta, \emptyset) \\ \textbf{push}(\Pi, \emptyset, pc, l) = \Pi' \qquad (\Theta, \beta, \Sigma, \Pi', \Lambda, P, l) \to (\Theta', \beta', \Sigma') \end{array}}{(\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc) \to (\Theta', \beta', \Sigma')}$$

$$(\textsc{Bf})\quad \frac{\begin{array}{c} P[pc] = \texttt{bz}\ v, l \qquad \textbf{split}(\Theta, \beta, v) = (\emptyset, \Theta) \\ \textbf{push}(\Pi, \emptyset, pc, l) = \Pi' \qquad (\Theta, \beta, \Sigma, \Pi', \Lambda, P, pc+1) \to (\Theta', \beta', \Sigma') \end{array}}{(\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc) \to (\Theta', \beta', \Sigma')}$$

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

$$(\textsc{Ba})\quad \frac{\begin{array}{c} P[pc] = \texttt{branch\_mask}\ T_{id}, l \\ T_{id} \in \Theta' \qquad (\Theta, \beta, \Sigma, \Pi, (\Theta', \Pi') : \Lambda, P, l) \to (\Theta'', \beta', \Sigma') \end{array}}{(\Theta, \beta, \Sigma, \Pi, (\Theta', \Pi') : \Lambda, P, pc) \to (\Theta'', \beta', \Sigma')}$$

$$(\textsc{Bi})\quad \frac{\begin{array}{c} P[pc] = \texttt{branch\_mask}\ T_{id}, l \\ T_{id} \notin \Theta' \qquad (\Theta, \beta, \Sigma, \Pi, (\Theta', \Pi') : \Lambda, P, pc+1) \to (\Theta'', \beta', \Sigma') \end{array}}{(\Theta, \beta, \Sigma, \Pi, (\Theta', \Pi') : \Lambda, P, pc) \to (\Theta'', \beta', \Sigma')}$$

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

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

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

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

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

$$(\textsc{It})\quad \frac{\begin{array}{c} P[pc] = \iota \\ \iota \notin \{\texttt{stop}, \texttt{bnz}, \texttt{bz}, \texttt{branch\_mask}, \texttt{sync}, \texttt{jump}, \texttt{everywhere}, \texttt{end\_everywhere}\} \\ (\Theta, \beta, \Sigma, \Theta_{mask}, \iota) \to (\beta', \Sigma') \\ (\Theta, \beta', \Sigma', \Pi, (\Theta_{mask}, \Pi') : \Lambda, pc+1) \to (\Theta', \beta'', \Sigma'') \end{array}}{(\Theta, \beta, \Sigma, \Pi, (\Theta_{mask}, \Pi') : \Lambda, P, pc) \to (\Theta', \beta'', \Sigma'')}$$

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:

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

$$(\textsc{Bt})\quad \frac{\begin{array}{c} P[pc] = \texttt{bz}\ v, l \qquad \textbf{split}(\Theta, \beta, v) = (\Theta, \emptyset) \\ \textbf{push}(\Pi, \emptyset, pc, l) = \Pi' \qquad (\Theta, \beta, \Sigma, \Pi', \Lambda, P, l) \rightarrow (\Theta', \beta', \Sigma') \end{array}}{(\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc) \rightarrow (\Theta', \beta', \Sigma')}$$

$$(\textsc{Bf})\quad \frac{\begin{array}{c} P[pc] = \texttt{bz}\ v, l \qquad \textbf{split}(\Theta, \beta, v) = (\emptyset, \Theta) \\ \textbf{push}(\Pi, \emptyset, pc, l) = \Pi' \qquad (\Theta, \beta, \Sigma, \Pi', \Lambda, P, pc+1) \rightarrow (\Theta', \beta', \Sigma') \end{array}}{(\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc) \rightarrow (\Theta', \beta', \Sigma')}$$

$$(\textsc{Bd})\quad \frac{\begin{array}{c} P[pc] = \texttt{bz}\ v, l \qquad \textbf{split}(\Theta, \beta, v) = (\Theta_0, \Theta_n) \\ \textbf{push}(\Pi, \Theta_n, pc, l) = \Pi' \qquad (\Theta_0, \beta, \Sigma, \Pi', \Lambda, P, pc+1) \rightarrow (\Theta', \beta', \Sigma') \end{array}}{(\Theta, \beta, \Sigma, \Pi, \Lambda, P, pc) \rightarrow (\Theta', \beta', \Sigma')}$$

$$(\textsc{Ba})\quad \frac{\begin{array}{c} P[pc] = \texttt{branch\_mask}\ T_{id}, l \\ T_{id} \in \Theta' \qquad (\Theta, \beta, \Sigma, \Pi, (\Theta', \Pi') : \Lambda, P, l) \rightarrow (\Theta'', \beta', \Sigma') \end{array}}{(\Theta, \beta, \Sigma, \Pi, (\Theta', \Pi') : \Lambda, P, pc) \rightarrow (\Theta'', \beta', \Sigma')}$$

$$(\textsc{Bi})\quad \frac{\begin{array}{c} P[pc] = \texttt{branch\_mask}\ T_{id}, l \\ T_{id} \notin \Theta' \qquad (\Theta, \beta, \Sigma, \Pi, (\Theta', \Pi') : \Lambda, P, pc+1) \rightarrow (\Theta'', \beta', \Sigma') \end{array}}{(\Theta, \beta, \Sigma, \Pi, (\Theta', \Pi') : \Lambda, P, pc) \rightarrow (\Theta'', \beta', \Sigma')}$$

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

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

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

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

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

$$(\textsc{It})\quad \frac{\begin{array}{c} P[pc] = \iota \\ \iota \notin \{\texttt{stop}, \texttt{bnz}, \texttt{bz}, \texttt{branch\_mask}, \texttt{sync}, \texttt{jump}, \texttt{everywhere}, \texttt{end\_everywhere}\} \\ (\Theta, \beta, \Sigma, \Theta_{mask}, \iota) \rightarrow (\beta', \Sigma') \\ (\Theta, \beta', \Sigma', \Pi, (\Theta_{mask}, \Pi') : \Lambda, pc+1) \rightarrow (\Theta', \beta'', \Sigma'') \end{array}}{(\Theta, \beta, \Sigma, \Pi, (\Theta_{mask}, \Pi') : \Lambda, P, pc) \rightarrow (\Theta', \beta'', \Sigma'')}$$

SWI Prolog

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

intel

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)
```

Function declaration: $f(p_1, \ldots, p_n)$;
Function call: $f(t_1\ a_1, \ldots, t_n\ a_n)$;
Function `extract`$(t^n, p^n, a^n, \mathrm{i})$
  for $k \in 1 \ldots n$ **do**
    if t_k == *uniform* **then**
      $p_k = a_k$;
    if t_k == *varying* **then**
      `shfl`(a_k, i);

	Instructions
	v0 = ↓tid
	v1 = (v0 == 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 = ↓v4
	v6 = v5 + 1
	↑v4 = v6
Next	v8 = v8 + 1
	v7 = (v8 == 4)
	bz v7, Loop
	end_everywhere
	↑tid = 1
Done	sync

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

Compilers Laboratory

Department of Computer Science
Universidade Federal de Minas Gerais
Federal University of Minas Gerais, Brazil

Function Call Re-Vectorization

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.

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**

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);
    }
  }
}
```

Too many **lines of code**

Warp-synchronous wrapper for SIMD memory copy.

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]);
}
```

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);
    }
  }
}
```

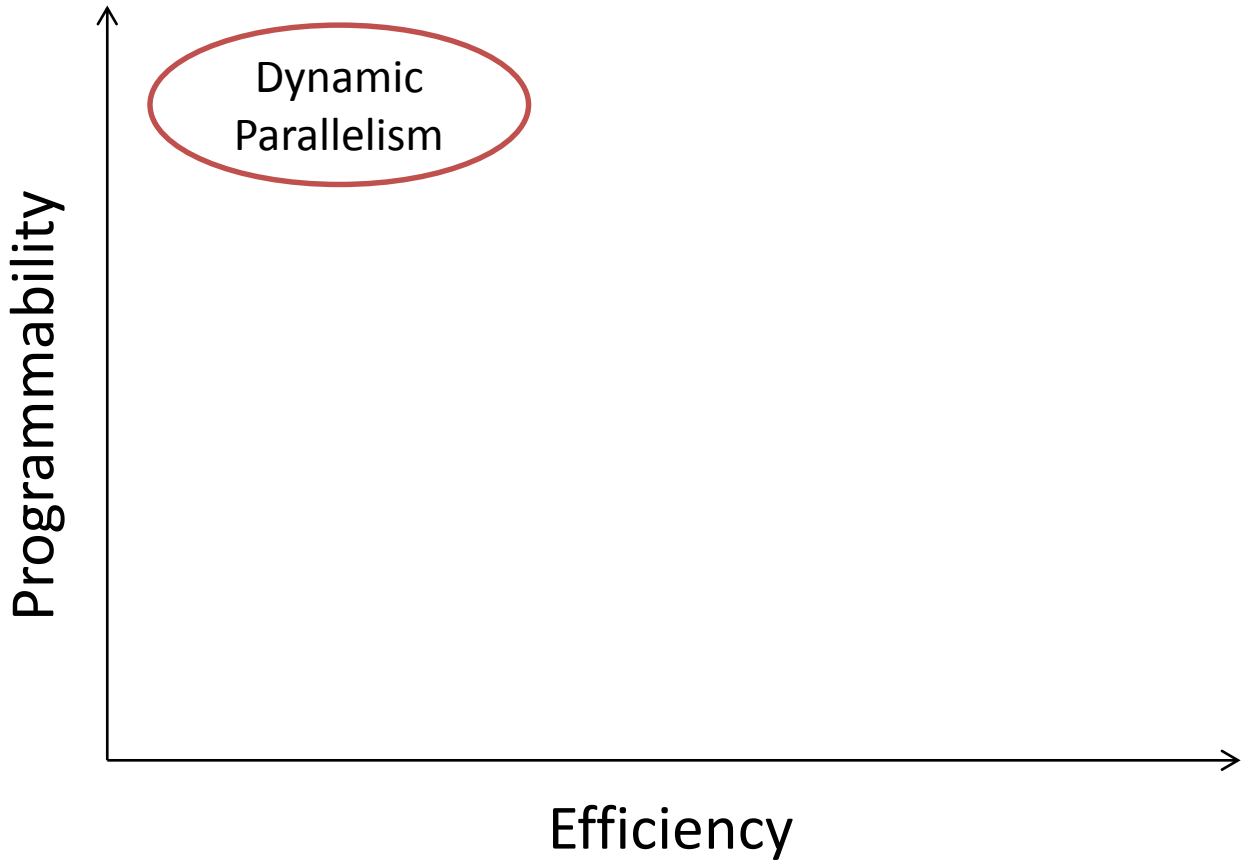
Too many **lines of code**

Warp-synchronous wrapper for SIMD memory copy.

Function Call Re-Vectorization: *Reprise*

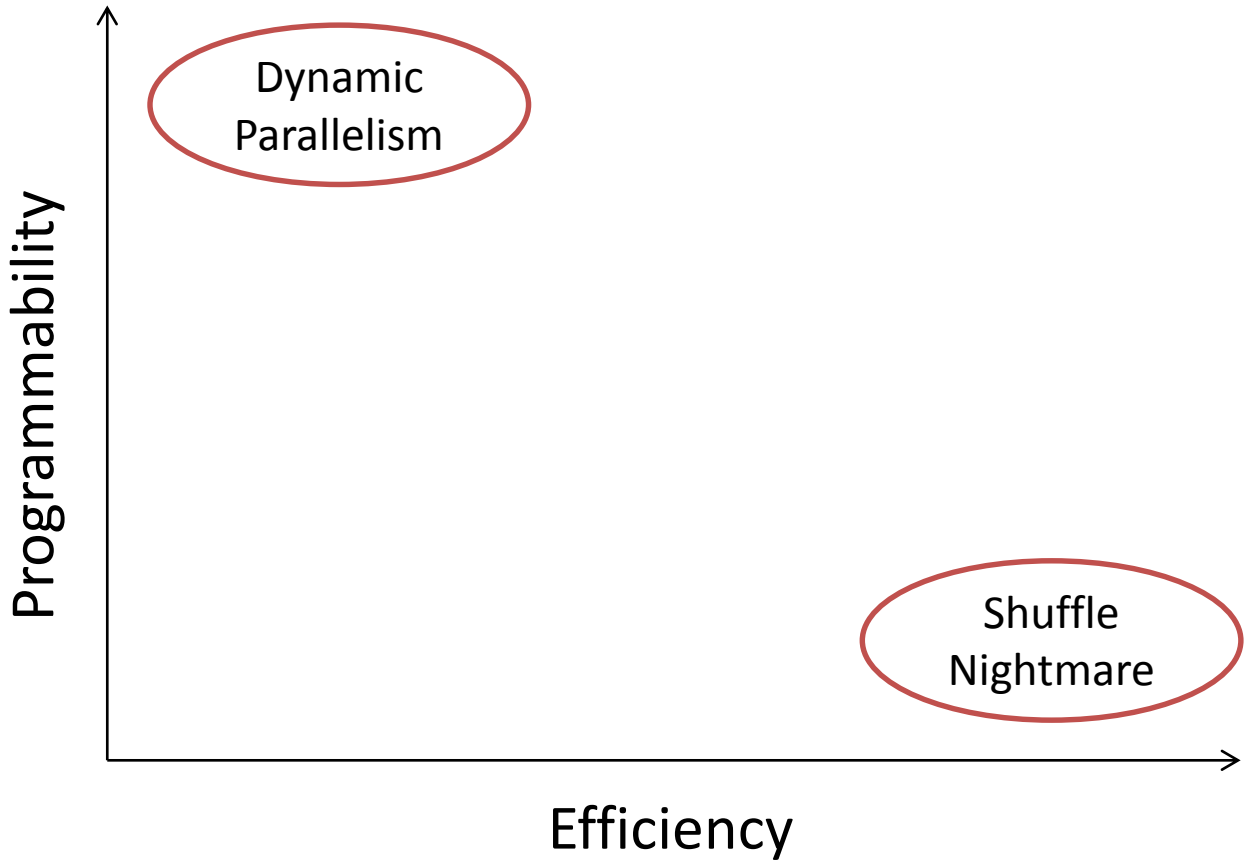
Function Call Re-Vectorization: *Reprise*

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



Function Call Re-Vectorization: *Reprise*

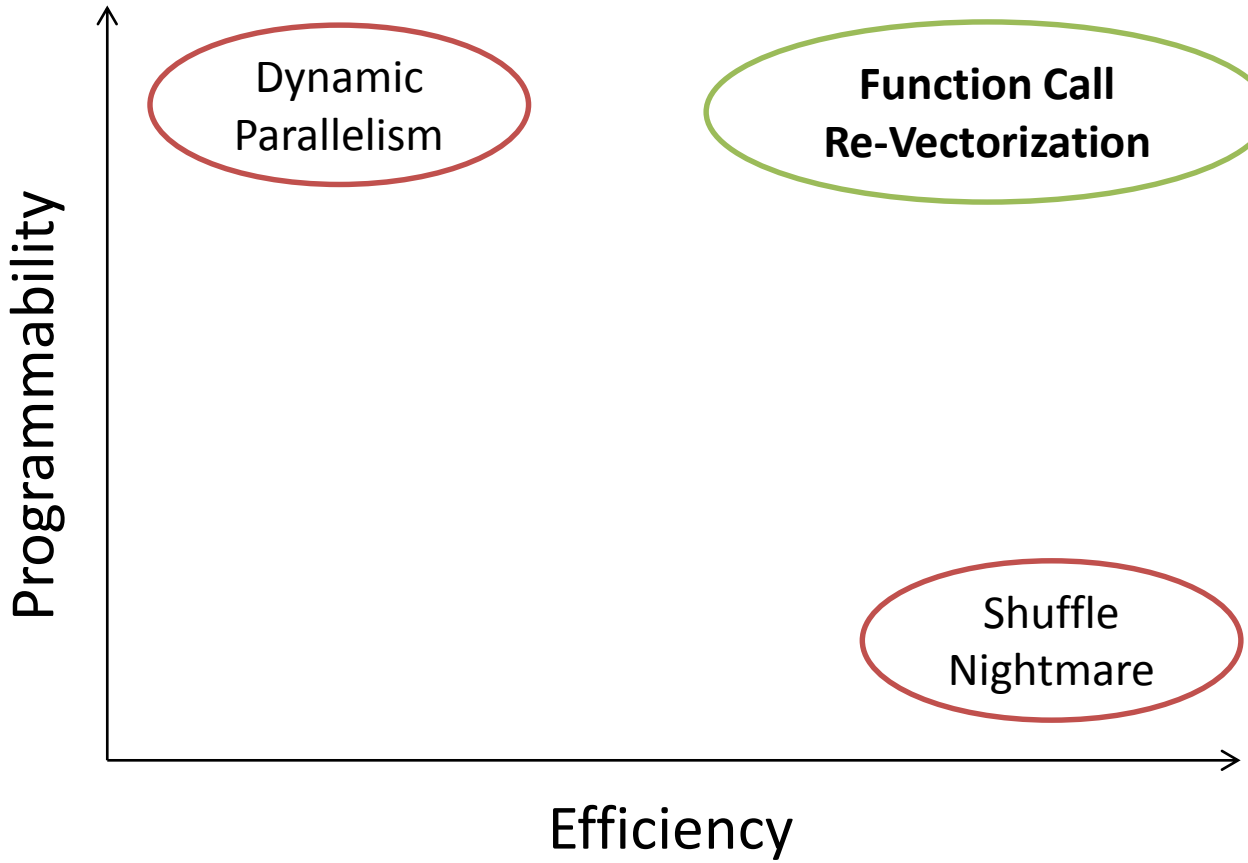
CUDA: kernel**<<<#warps, #threads>>>**(args...)



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

Function Call Re-Vectorization: *Reprise*

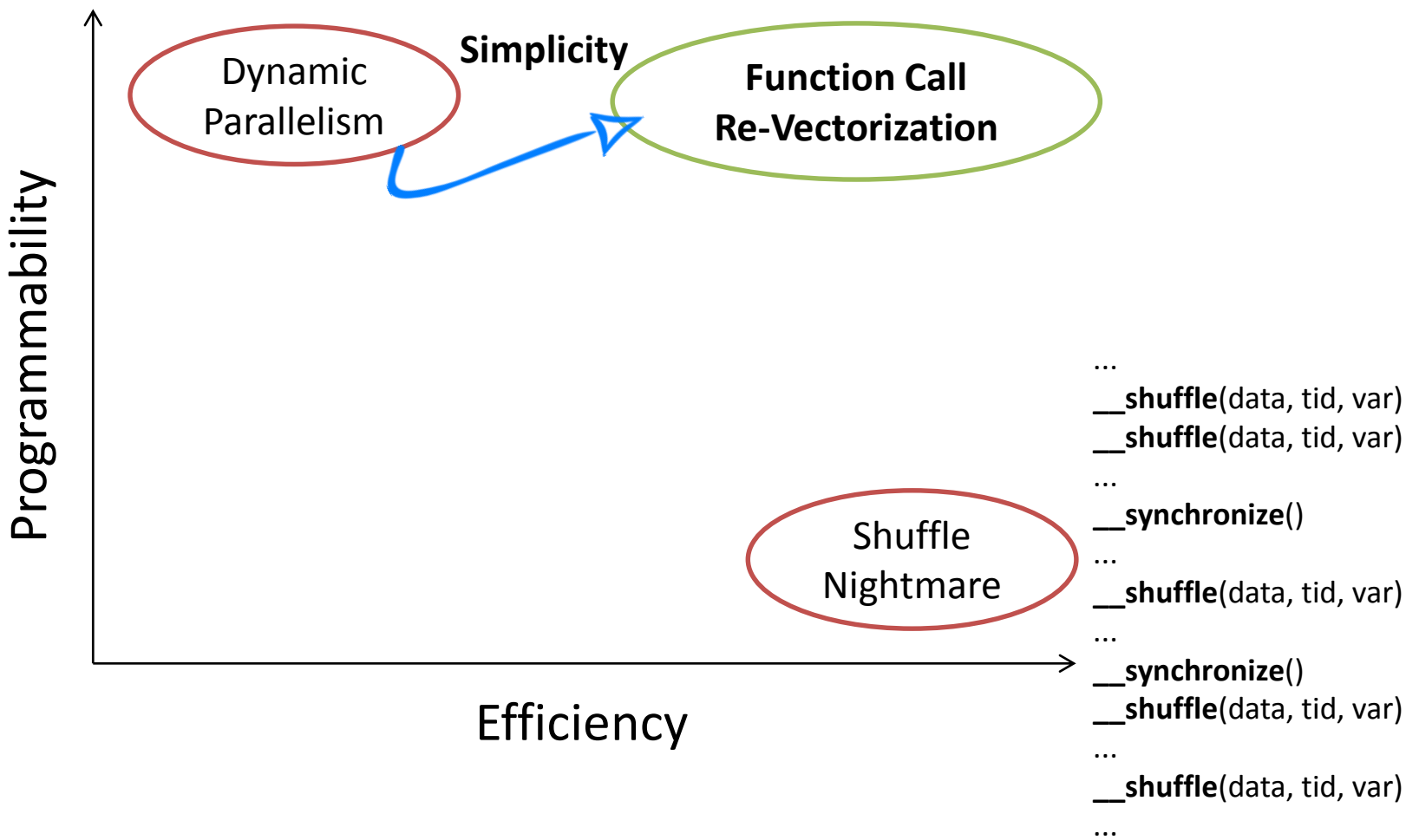
CUDA: kernel**<<<#warps, #threads>>>**(args...)



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

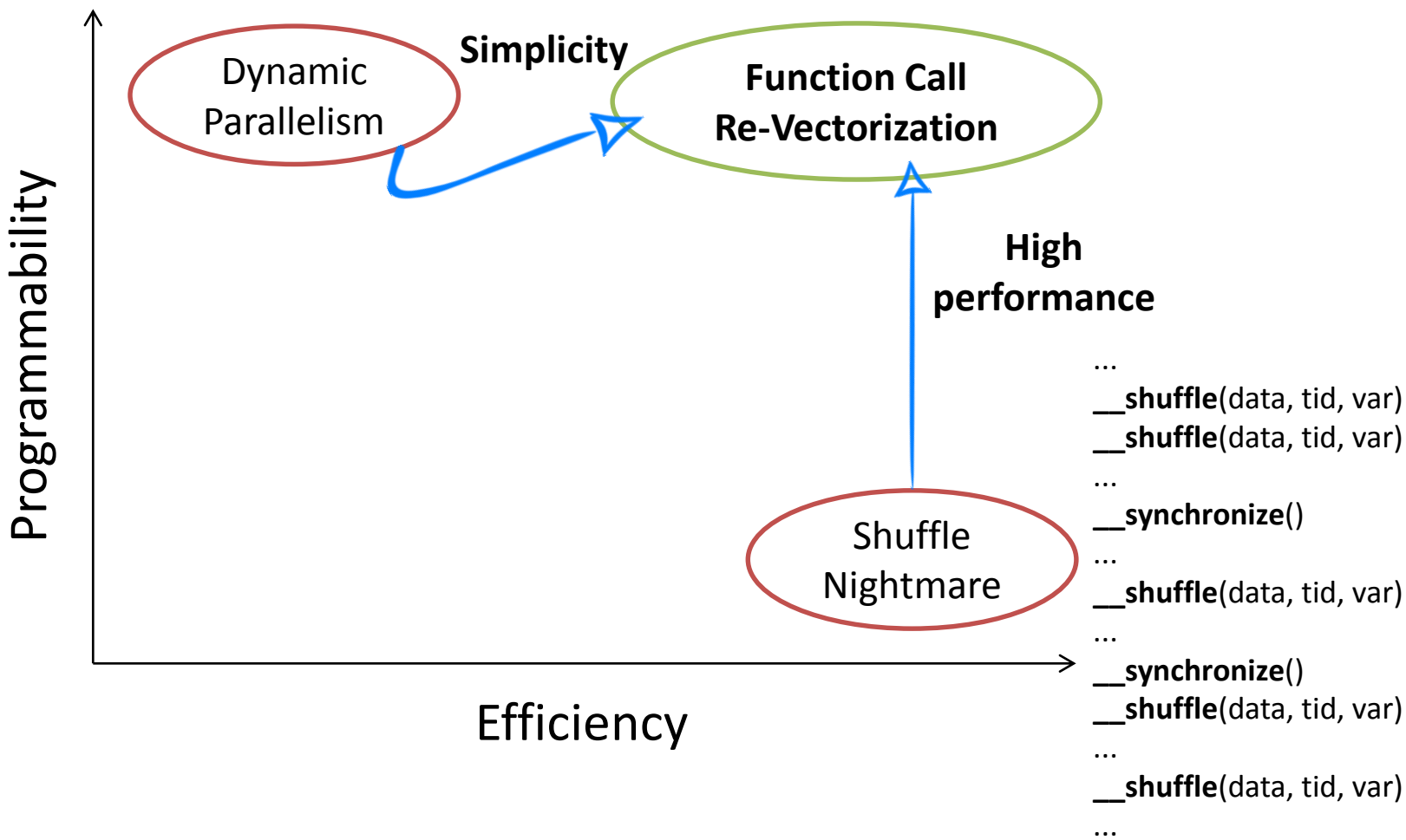
Function Call Re-Vectorization: *Reprise*

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



Function Call Re-Vectorization: *Reprise*

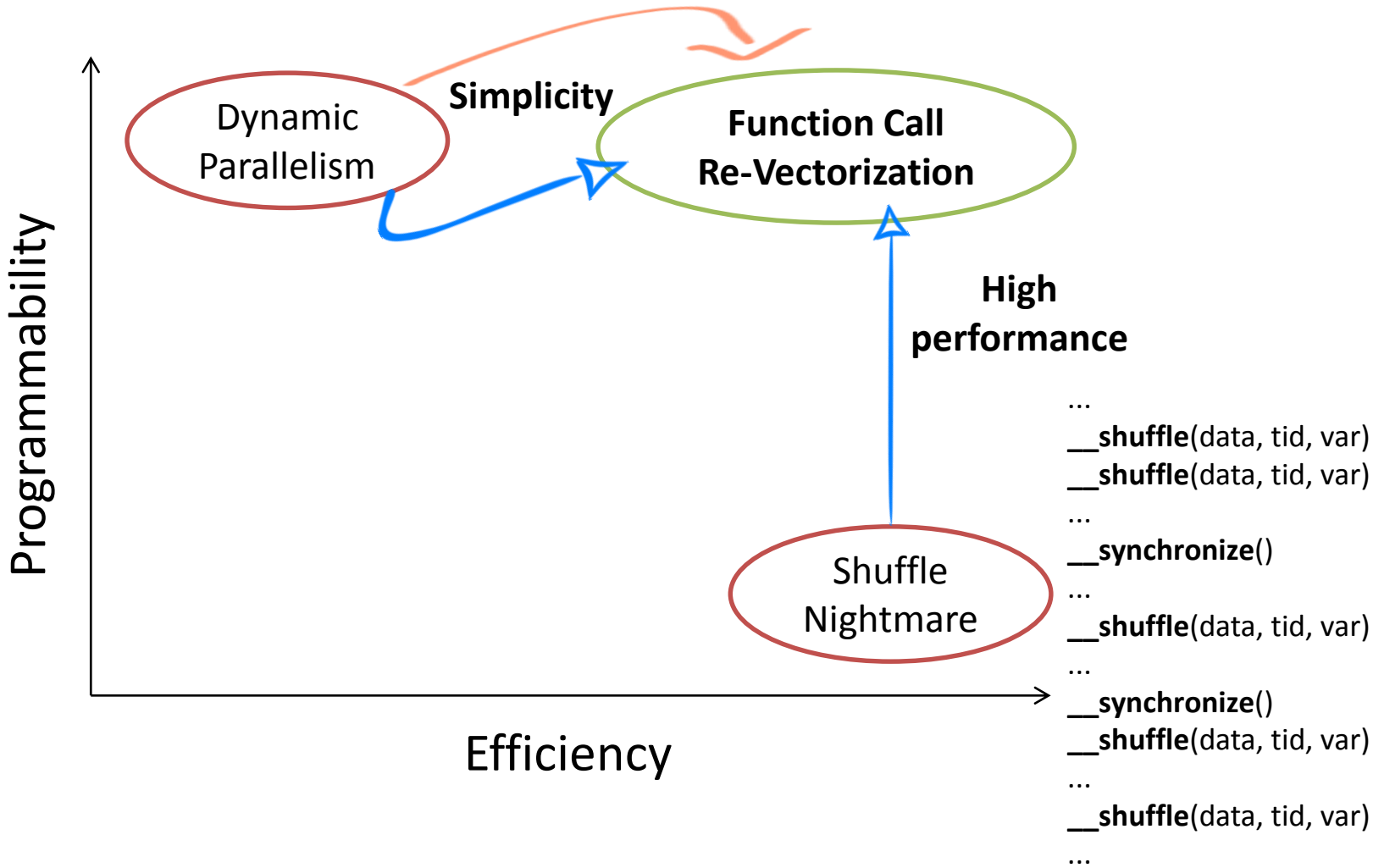
CUDA: kernel**<<<#warps, #threads>>>**(args...)



Function Call Re-Vectorization: *Reprise*

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

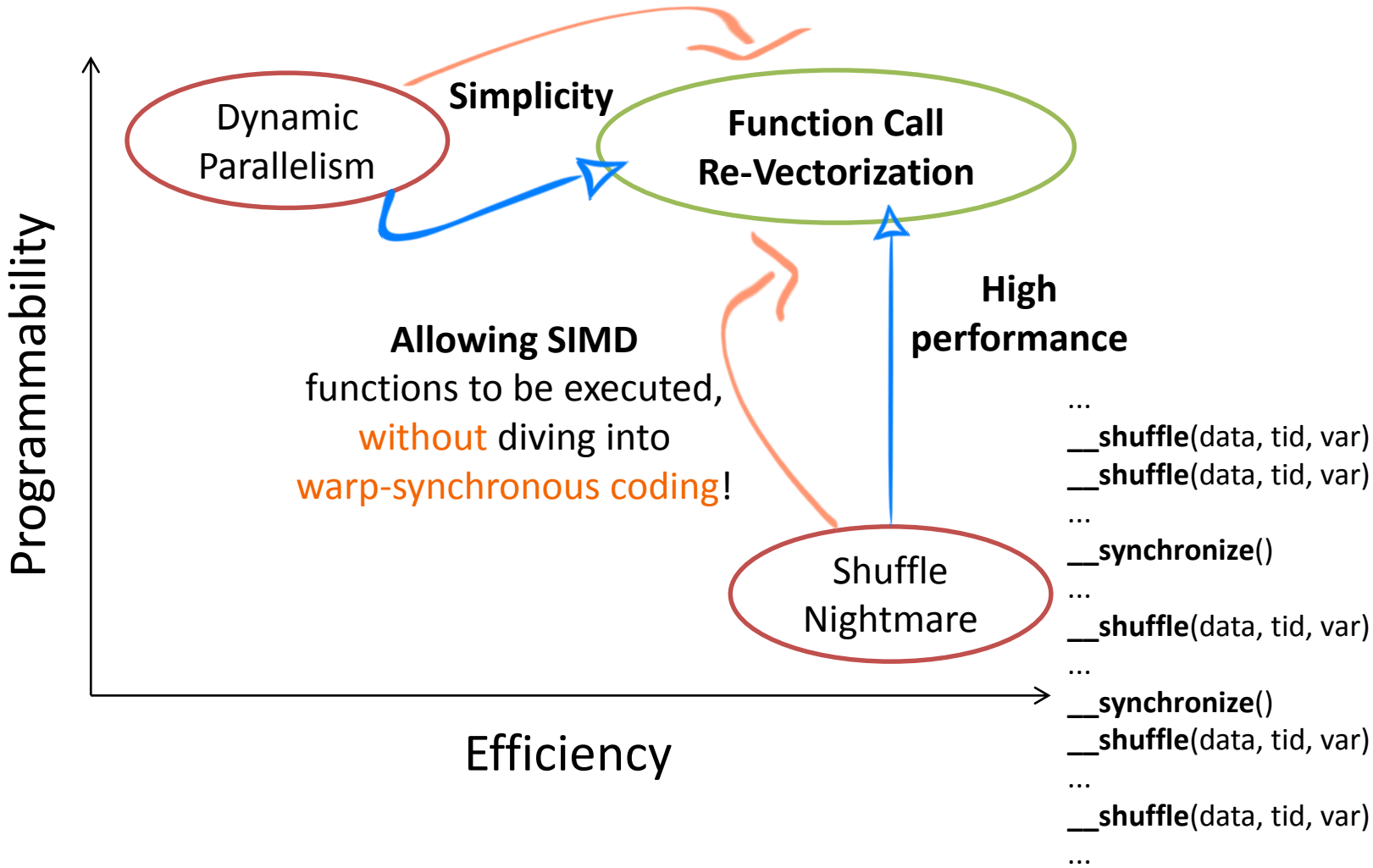
Re-enable all threads within warp,
avoiding kernel allocation and scheduling



Function Call Re-Vectorization: *Reprise*

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

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



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/CUDADownloads/TechBrief_Dynamic_Parallelism_in_CUDA.pdf

Function Call Re-Vectorization: Properties

Multiplicative composition

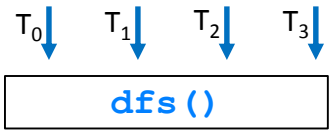
The target **crev** function runs once per active thread.

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

If the call is recursive, up to $\mathbf{W^N}$ 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);
        }

    }

}
```

Vectorized!



n = 1

Function Call Re-Vectorization: Properties

Multiplicative composition

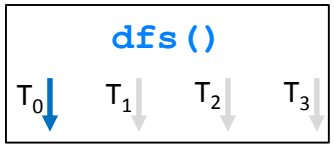
The target **crev** function runs once per active thread.

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

If the call is recursive, up to $\mathbf{W^N}$ 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);
        }

    }

}
```

Vectorized!



n = 1

Function Call Re-Vectorization: Properties

Multiplicative composition

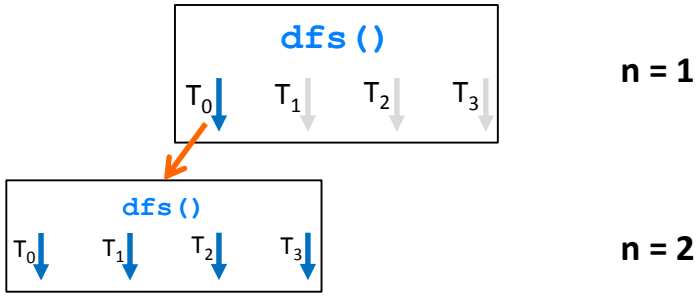
The target **crev** function runs once per active thread.

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

If the call is recursive, up to $\mathbf{W^N}$ 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);
        }

    }

}
```

Vectorized!



Function Call Re-Vectorization: Properties

Multiplicative composition

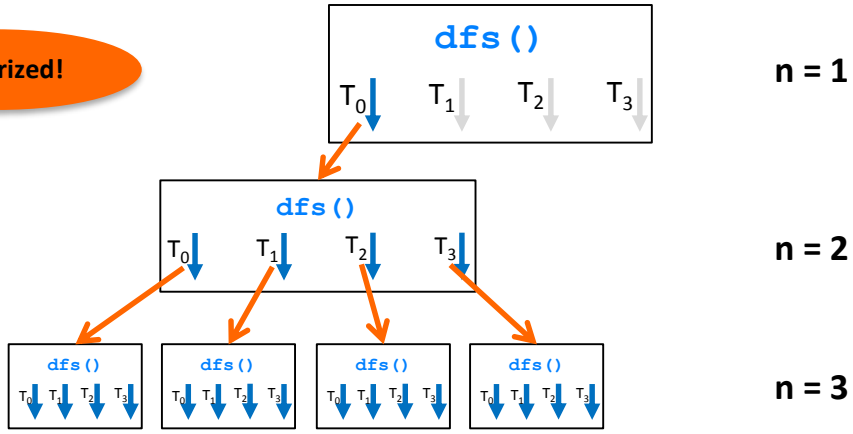
The target **crev** function runs once per active thread.

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

If the call is recursive, up to $\mathbf{W^N}$ 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);
        }

    }

}
```

Vectorized!



Function Call Re-Vectorization: Properties

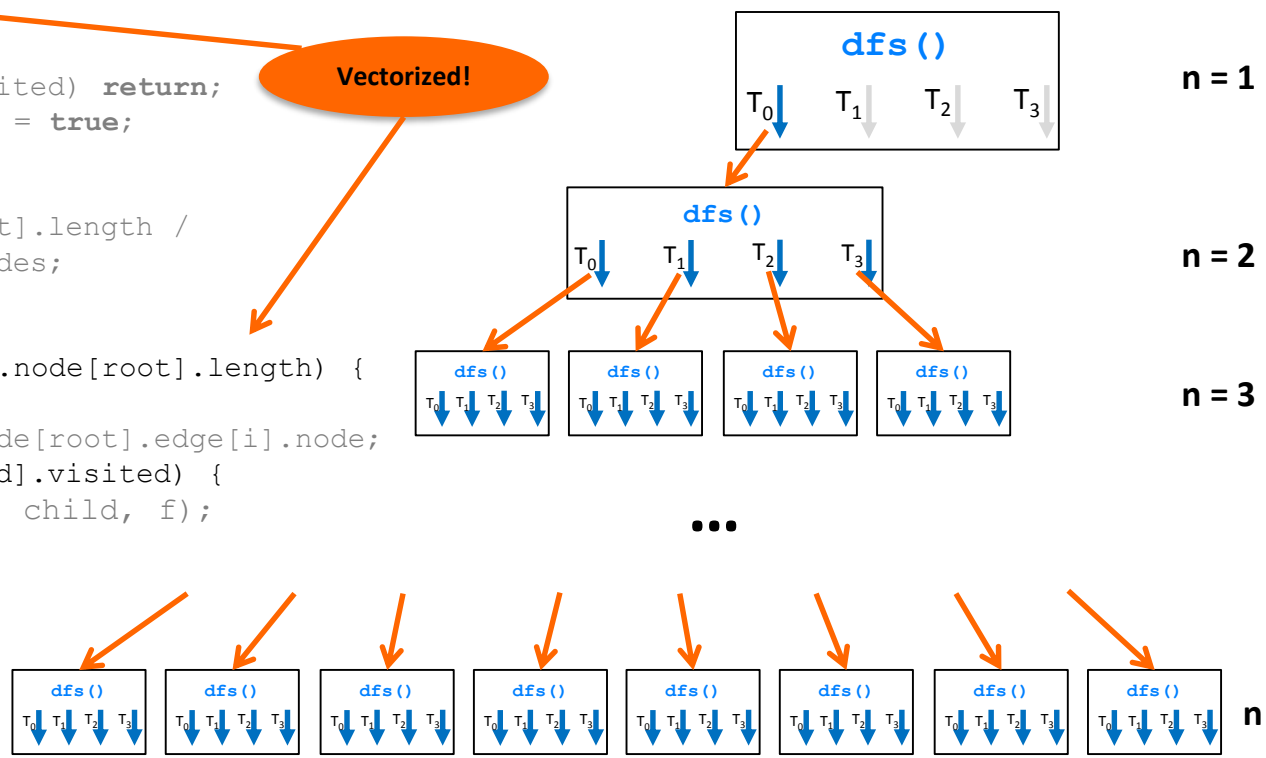
Multiplicative composition

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 $\mathbf{W^N}$ 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);
        }

    }

}
```

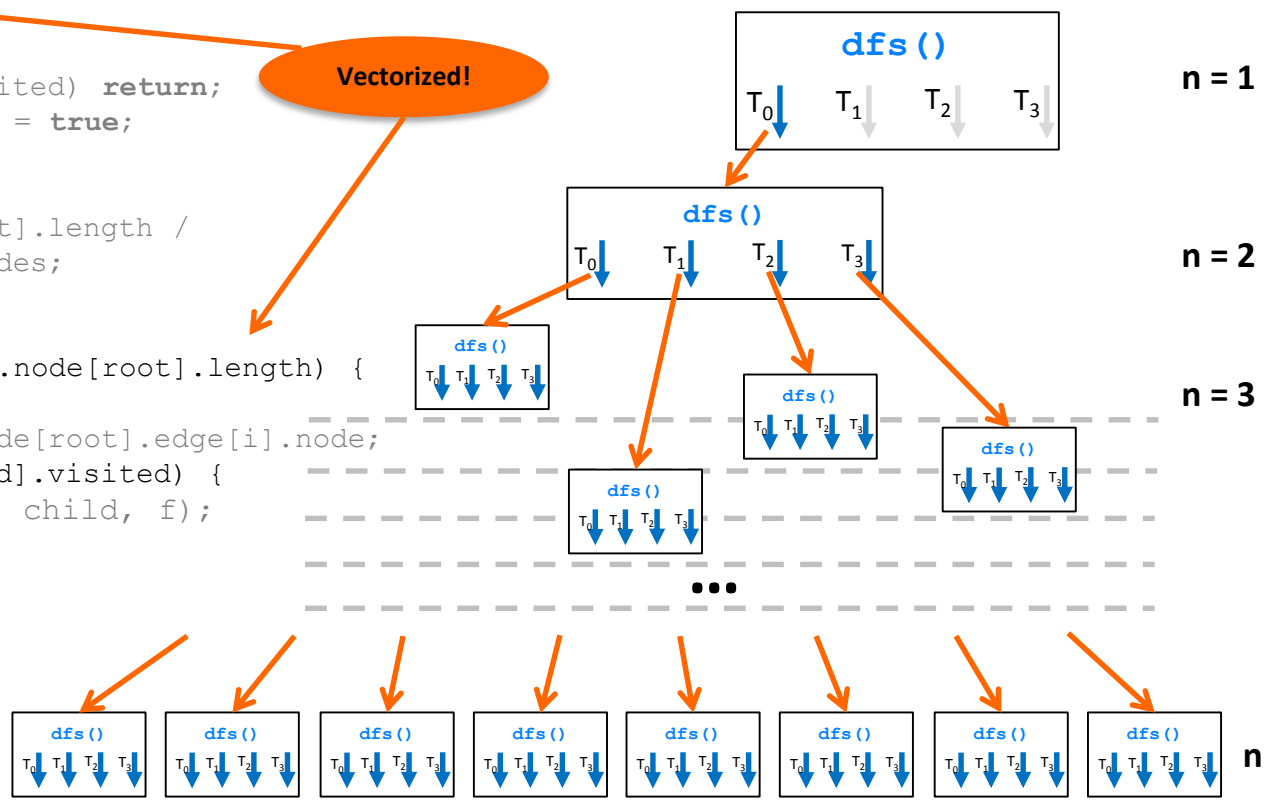


Function Call Re-Vectorization: Properties

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);
        }

    }

}
```



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)
...
```


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

Compilers Laboratory

Department of Computer Science
Universidade Federal de Minas Gerais
Federal University of Minas Gerais, Brazil

Experimental Evaluation

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.

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)

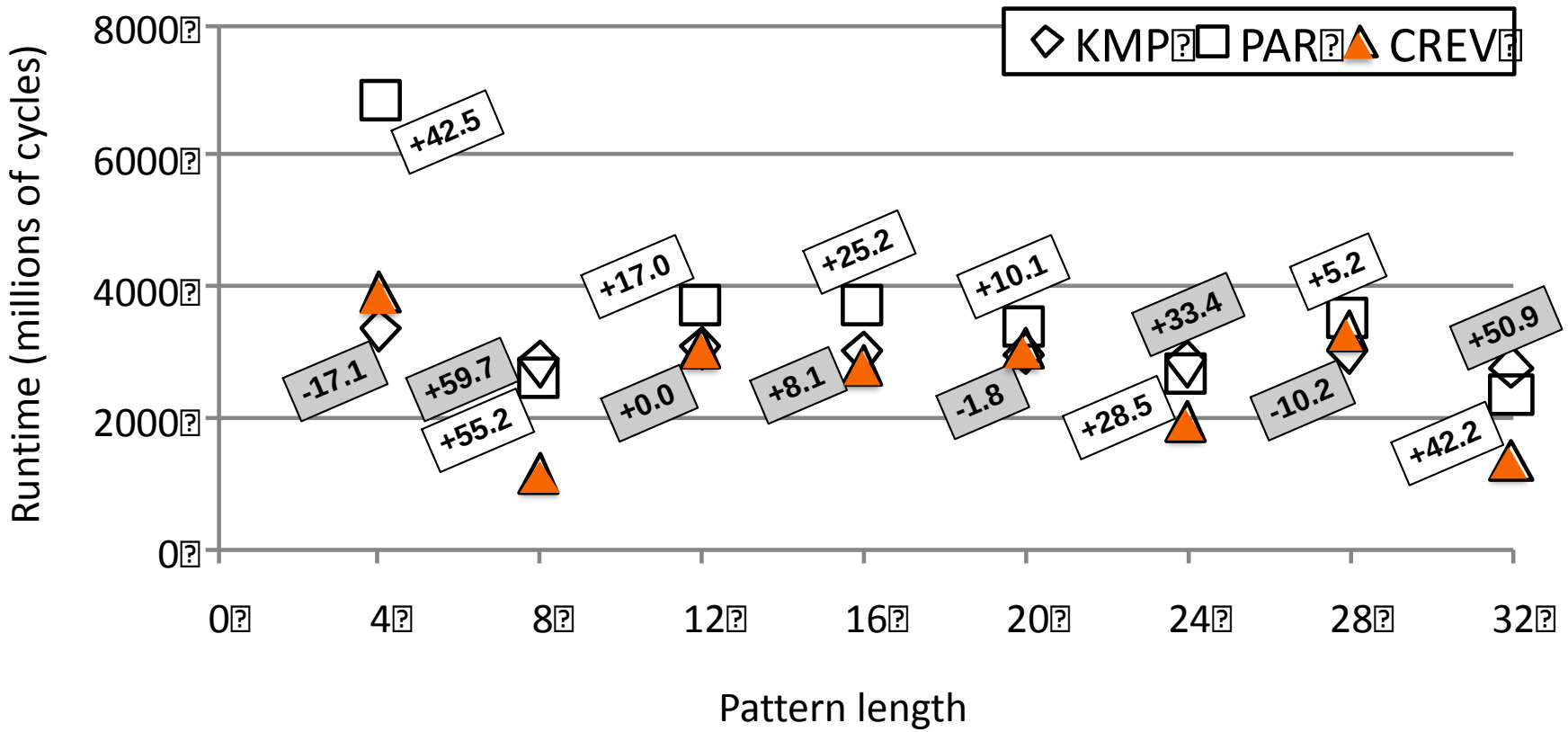
Benchmarks: seven algorithms implemented in different ways:

- **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; ✪ 1st runner up; ✪ 2nd runner up.

	Sequential	Parallel	Launch	CREV	Dataset
BookFilter	-- not implemented	8530.990	7857.980	7405.175	*bin-L20K-P16*
String Matching	6649.279 KMP Algorithm	3576.143	393166.268	2737.939	*txt-256MB-P16*
Bellman-Ford	141088.730	493619.688	-- thread lim exc	529856.065	*erdos-renyi*
Depth-First Search	3754.101	3786.263	-- thread lim exc	3790.444	*octree-D5*
Connected-Component Leader	4054.658	3983.088	5272.919	3984.795	*octree-D5*
Quicksort-bitonic	2.871	-- not implemented	204.278	2.878	*int-16K*
Mergesort-bitonic	7.302	-- 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; ✪ 1st runner up; ✪ 2nd runner up.

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

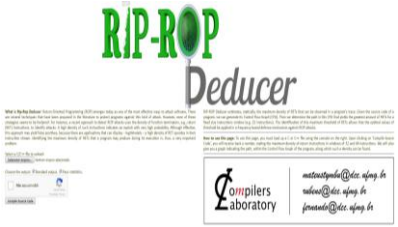
Contributions

Contributions: Rat

Paper published in CGO'16

Inference of Peak Density of Indirect Branches to Detect ROP Attacks

ACM CGO-SRC'16 winner
(1st place: golden medal)



Webpage with static analysis available

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

Paper published in SBSEG'15

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

Artifact Evaluated paper benchmarks

Contributions: Swan

Paper published in PPoPP'17

Function Call Re-Vectorization

Webpage with all source code and results available

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

Artifact Evaluated paper benchmarks

function call revectorization

Paper published in SBLP'16

Definição Semântica de Blocos Everywhere para Programação SIMD

SWI Prolog

Semantics of everywhere blocks in SIMD context

µSIMD Prolog abstract machine

Compilers Laboratory

Department of Computer Science
Universidade Federal de Minas Gerais
Federal University of Minas Gerais, Brazil

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

Questions?

Email us:
Rubens Emilio Alves Moreira [rubens@dcc.ufmg.br]
Fernando Magno Quintão Pereira [fernando@dcc.ufmg.br]

Check our websites:
http://cuda.dcc.ufmg.br/swan
http://cuda.dcc.ufmg.br/rip-rop-deducer