Compilers Laboratory

DCC
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

ICEx
Instituto de Ciências Exatas

UFMG

Everywhere Blocks for SIMD Programming

Authors: Rubens E. A. Moreira, Sylvain Collange, Fernando M. Q. Pereira

Speaker: Breno Campos Ferreira Guimarães

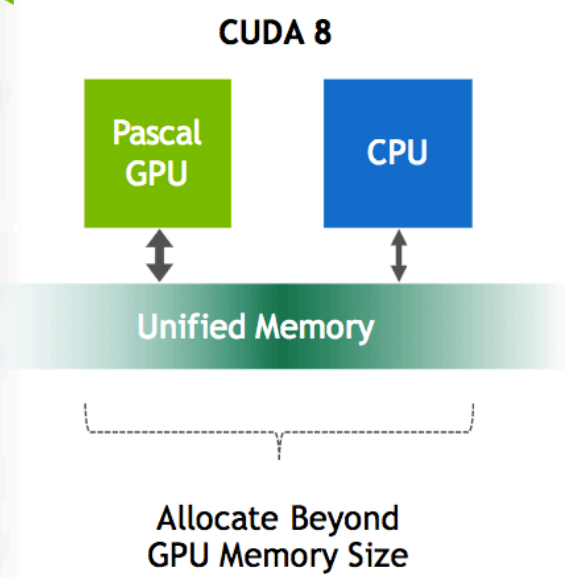
CUDA 8 AND BEYOND

Mark Harris, April 5, 2016

GPU TECHNOLOGY CONFERENCE

CUDA 8: UNIFIED MEMORY

Large datasets, simple programming, High Performance



Enable Large Data Models	Oversubscribe GPU memory Allocate up to system memory size
Simpler Data Accesss	CPU/GPU Data coherence Unified memory atomic operations
Tune Unified Memory Performance	Usage hints via cudaMemAdvise API Explicit prefetching API

8 NVIDIA

Simple *andalso* efficient

Source: http://on-demand.gputechconf.com/gtc/2016/presentation/s6224-mark-harris.pdf

CUDA 8 AND BEYOND

Mark Harris, April 5, 2016

GPU TECHNOLOGY CONFERENCE

MOTIVATING EXAMPLE

Safe, Explicit Programming for Performance

Approximately equal performance to unsafe warp programming

```
__device__
int warp_reduce(int val) {
  extern __shared__ int smem[];
  const int tid = threadIdx.x;

  #pragma unroll
  for (int i = warpSize/2; i > 0; i /= 2) {
    smem[tid] = val;        sync(this_warp());
    val += smem[tid ^ i];   sync(this_warp());
  }
  return val;
}
```

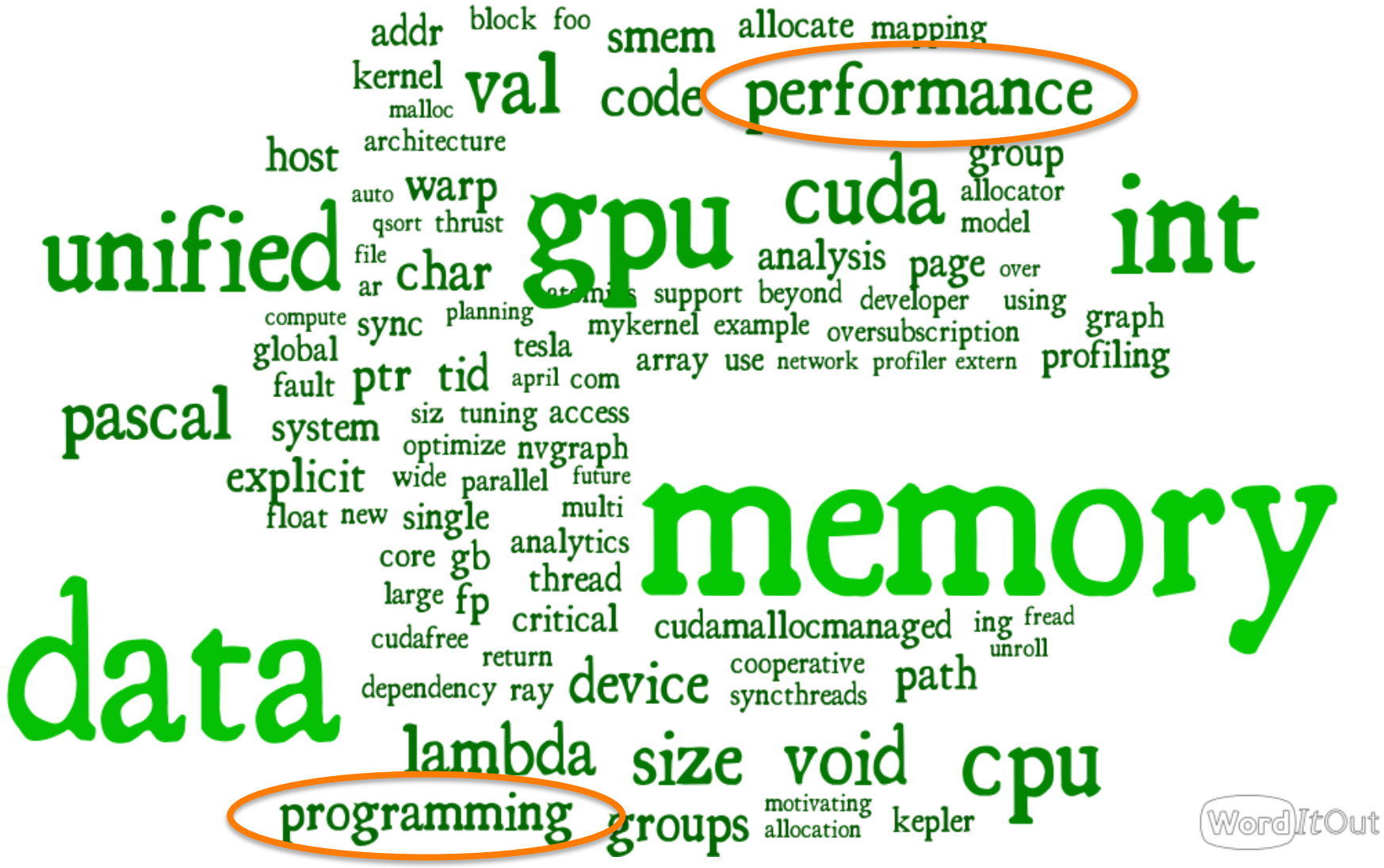
Explicit, *yet safe* programming!

Safe *and* Fast!

37 NVIDIA

Source: http://on-demand.gputechconf.com/gtc/2016/presentation/s6224-mark-harris.pdf

Trends in Massively Parallel Processing



Source: http://on-demand.gputechconf.com/gtc/2016/presentation/s6224-mark-harris.pdf

Compilers Laboratory

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

Divergences

Divergences

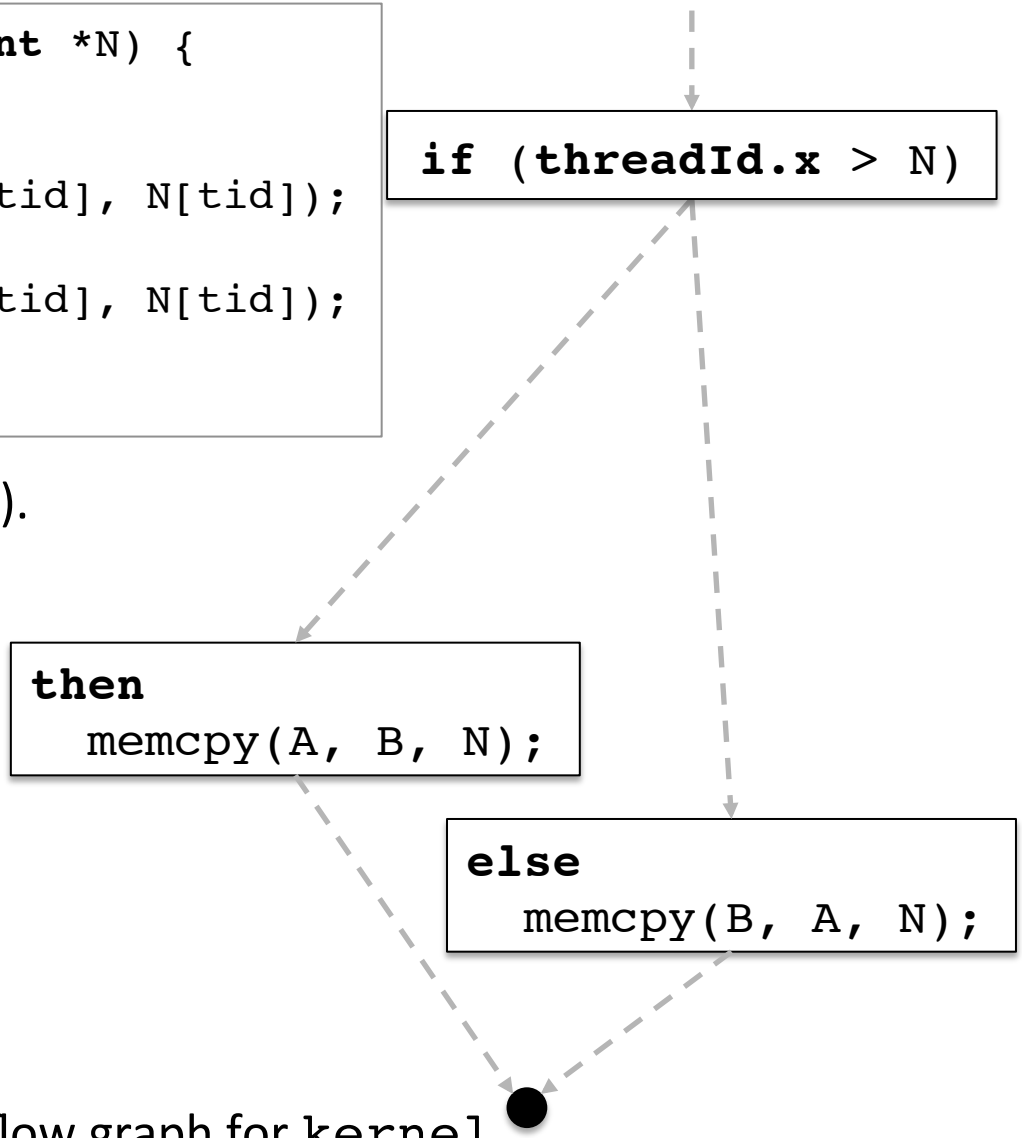
```
void kernel(int **A, int **B, int *N) {
  int tid(threadId.x);
  if (tid > N) {
    memcpy<<<1, 4>>>(A[tid], B[tid], N[tid]);
  } else {
    memcpy<<<1, 4>>>(B[tid], A[tid], N[tid]);
  }
}
```

Kernel for parallel execution (CUDA).

Divergences

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

Kernel for parallel execution (CUDA).

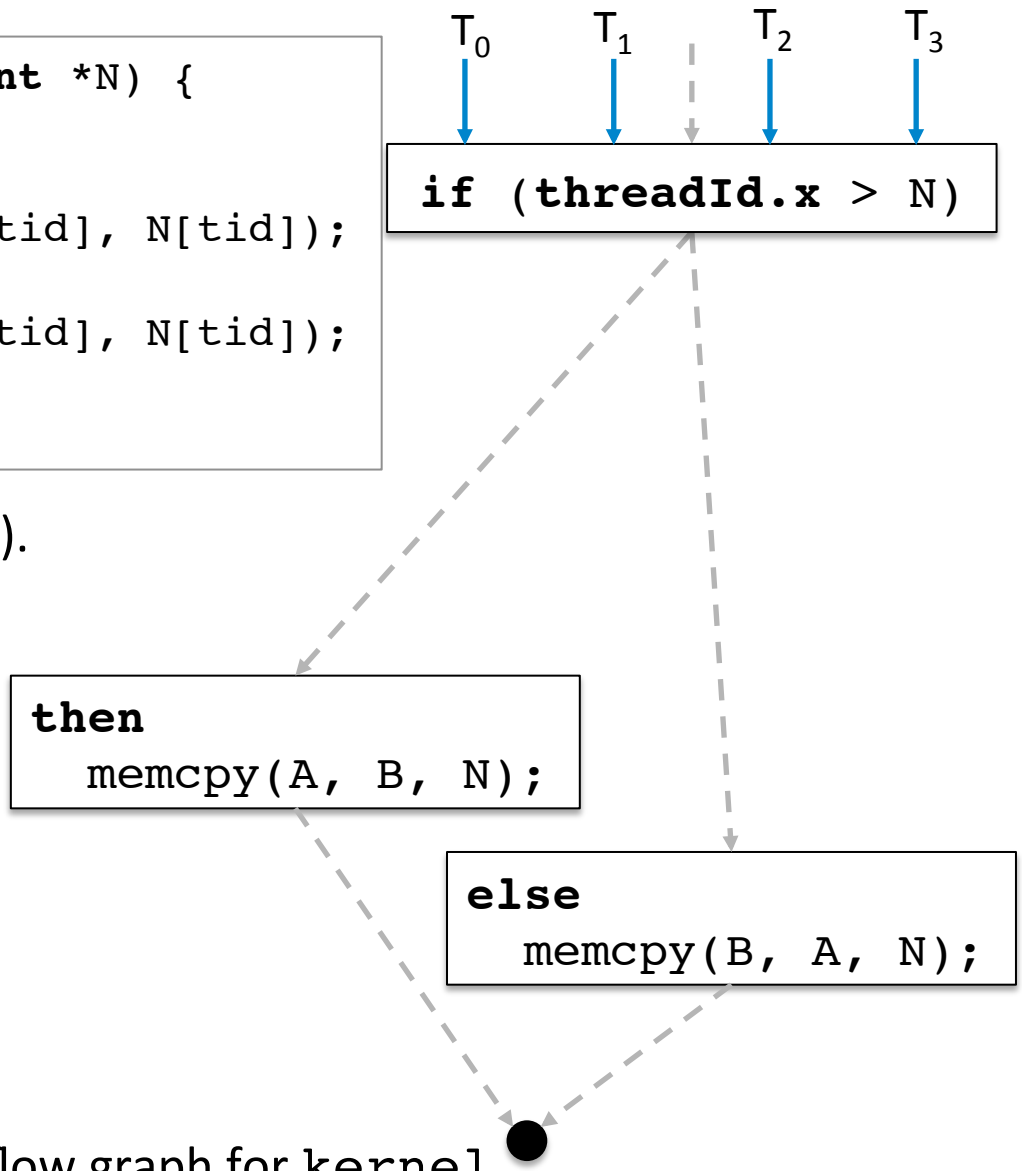


Control flow graph for `kernel`.

Divergences

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

Kernel for parallel execution (CUDA).



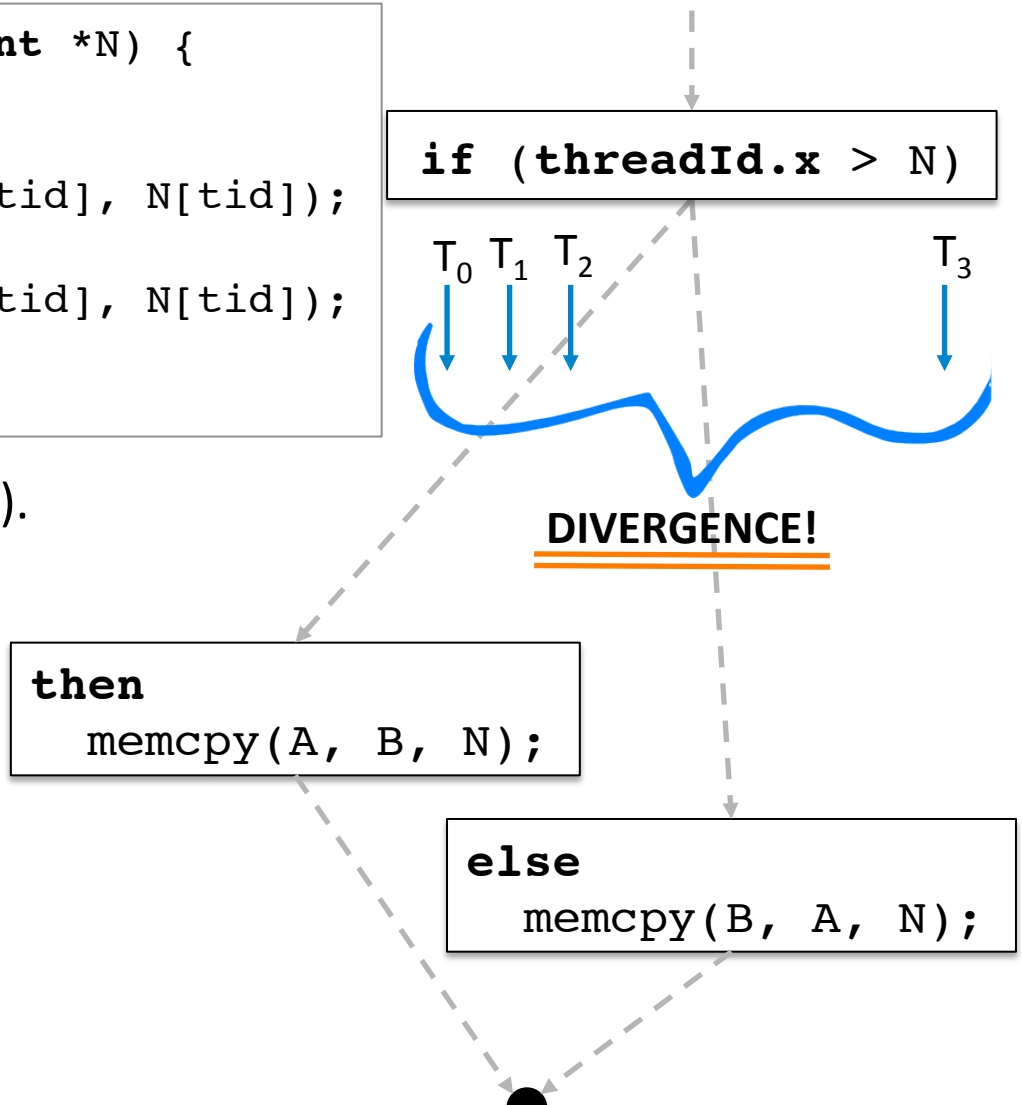
SIMD: LOCKSTEP EXECUTION!

Control flow graph for `kernel`.

Divergences

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

Kernel for parallel execution (CUDA).



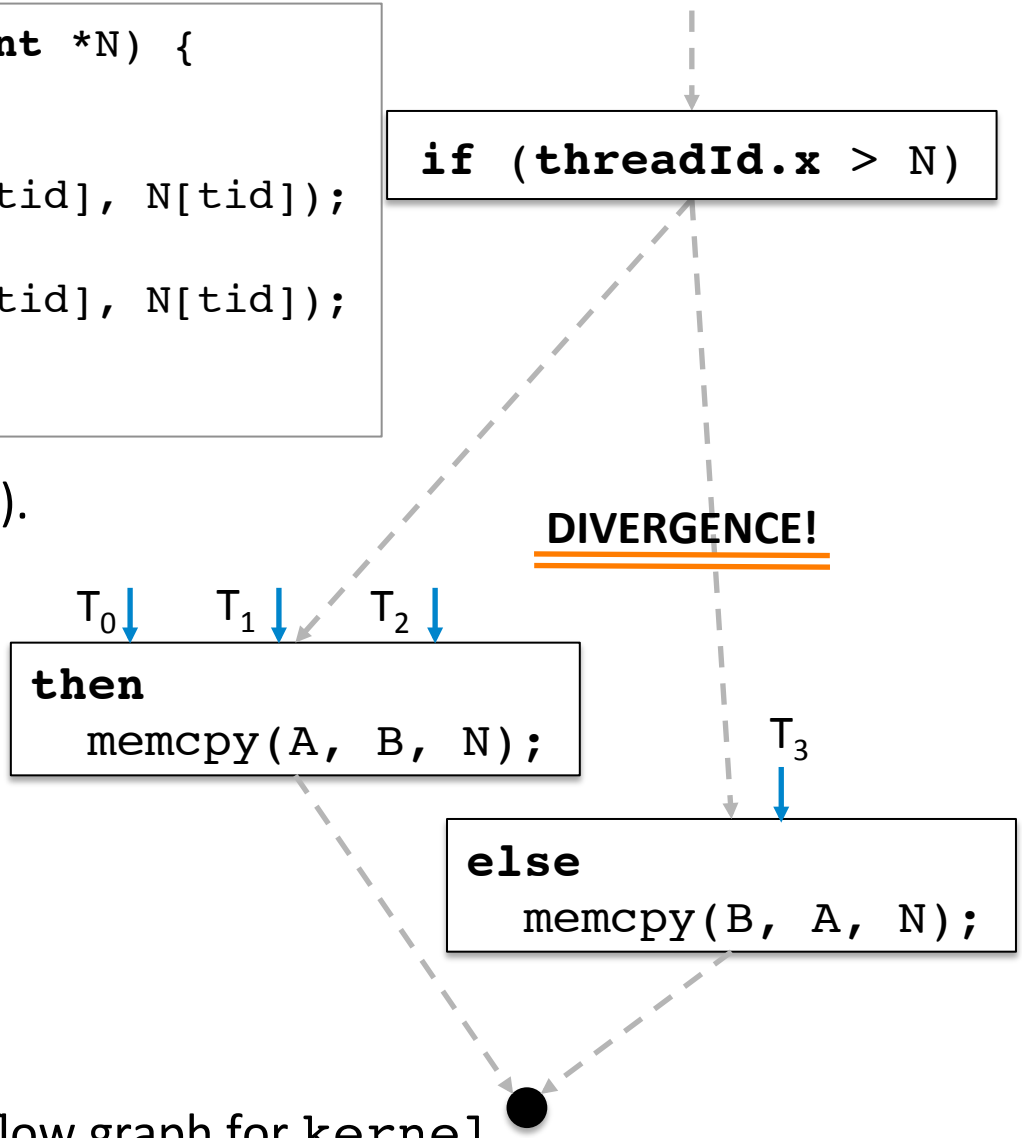
SIMD: LOCKSTEP EXECUTION!

Control flow graph for `kernel`.

Divergences

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

Kernel for parallel execution (CUDA).



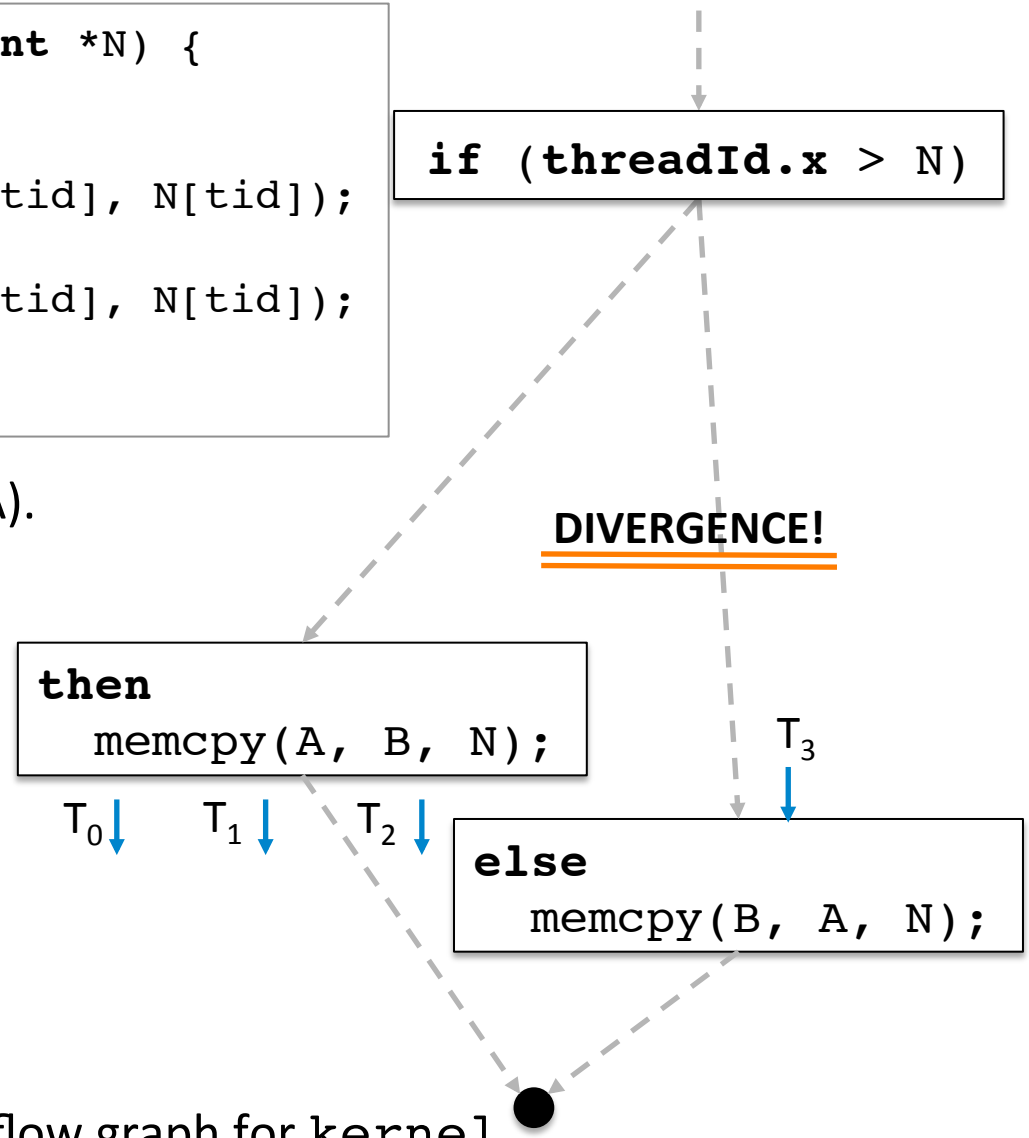
SIMD: LOCKSTEP EXECUTION!

Control flow graph for `kernel`.

Divergences

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

Kernel for parallel execution (CUDA).



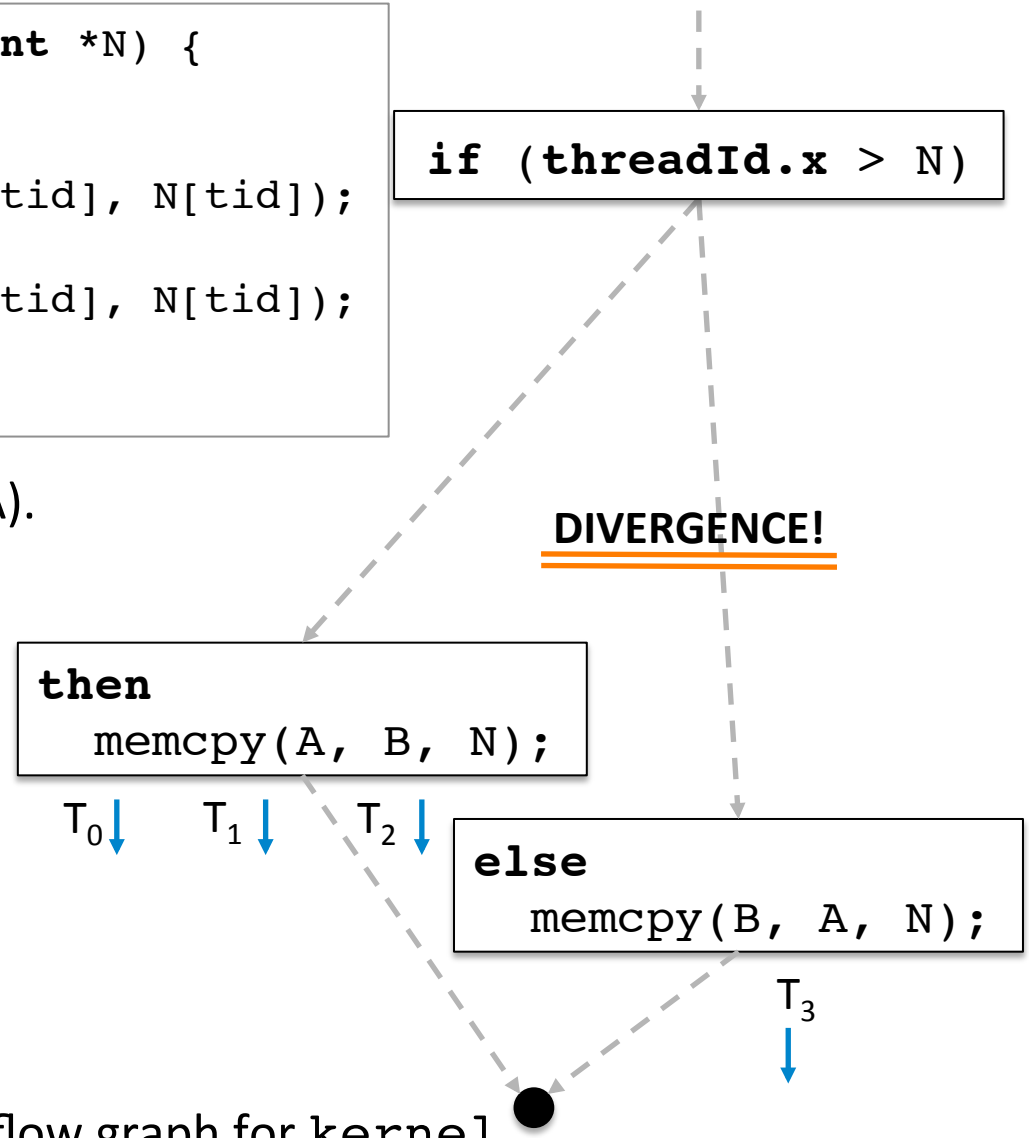
SIMD: LOCKSTEP EXECUTION!

Control flow graph for `kernel`.

Divergences

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

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 > N) {
    memcpy<<<1, 4>>>(A[tid], B[tid], N[tid]);
  } else {
    memcpy<<<1, 4>>>(B[tid]
  }
}
```

Kernel for pa

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

ERGENCE!

```
if (threadId.x > N)
```

```
  memcpy(A, B, N);
```

T_0 T_1 T_2

```
else
  memcpy(B, A, N);
```

T_3

SIMD: LOCKSTEP EXECUTION!

Control flow graph for `kernel`.

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

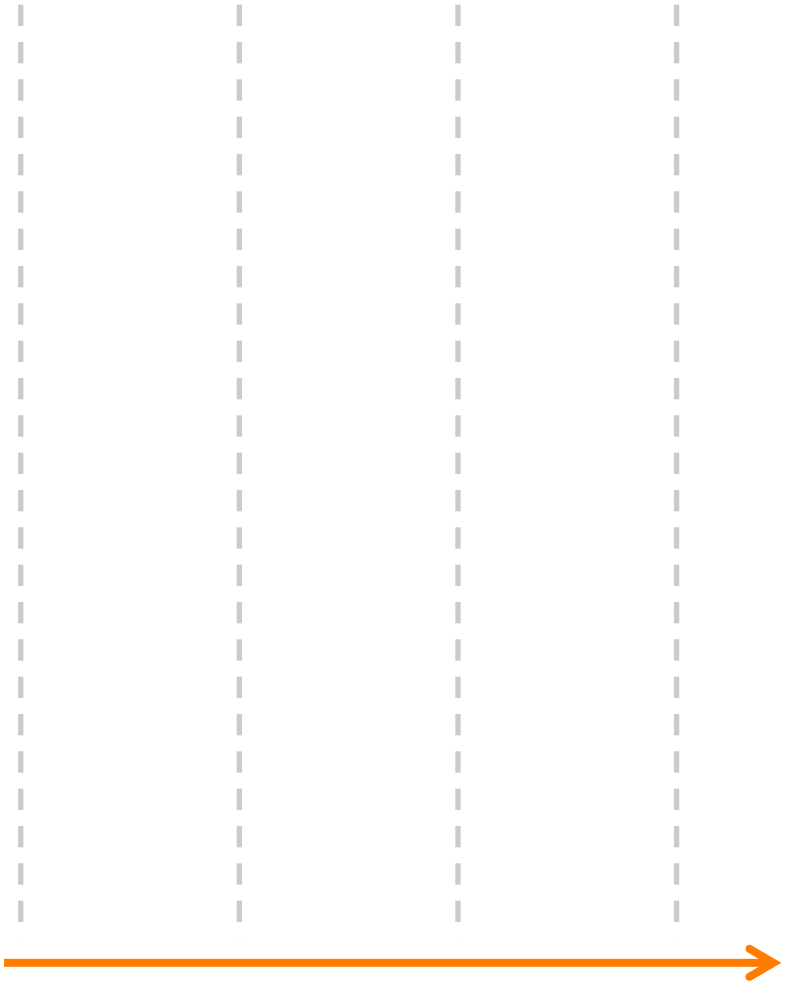
```
int idx = threadId.x;
int dimx = threadDim.x;
```

```
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

```
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;
  }
}
```

16153μs:
constant assignment

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;
  }
}
```

16250μs:
few operations and assignment

16153μs:
constant assignment

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;
  }
}
```

16250μs:
few operations and assignment

16153μs:
constant assignment

30210μs:
function call and assignment

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;
  }
}
```

16250μs:
few operations and assignment

16153μs:
constant assignment

32193μs:
constant assignment BUT <u>within divergent region</u>!

30210μs:
function call and assignment

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

Interlude: The Kernels of Samuel

Divergence is harmful to performance!

```
int id
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: few operations and assignment

16153μs: constant assignment

32193μs: constant assignment **BUT** <u>within divergent region</u>!

30210μs: function call and assignment

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 > N) {
    memcpy<<<1, 4>>>(A[tid], B[tid], N[tid]);
  } else {
    memcpy<<<1, 4>>>(B[tid], A[tid], N[tid]);
  }
}
```

Kernel for parallel execution (CUDA).

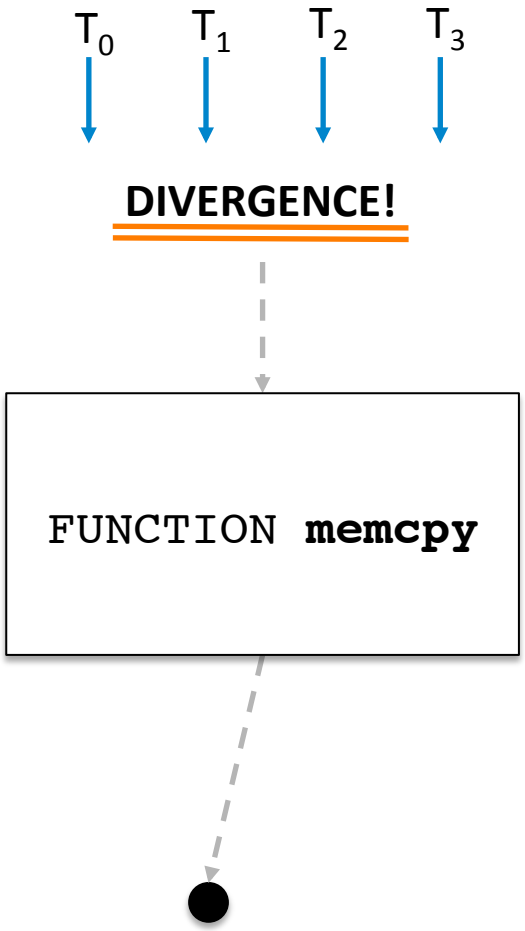
Divergent region:
only active threads
run **memcpy**

Divergences: Coda

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

Kernel for parallel execution (CUDA).

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



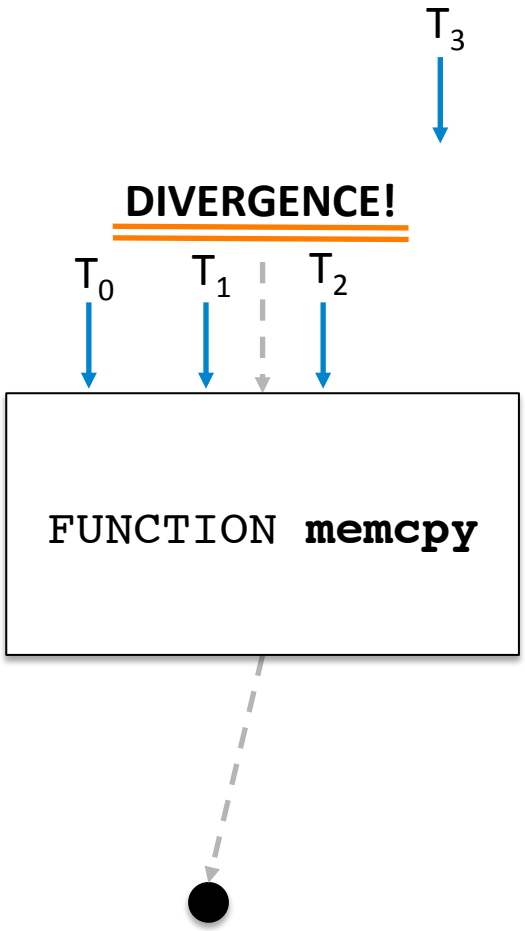
Control flow graph for memcpy.

Divergences: Coda

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

Kernel for parallel execution (CUDA).

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



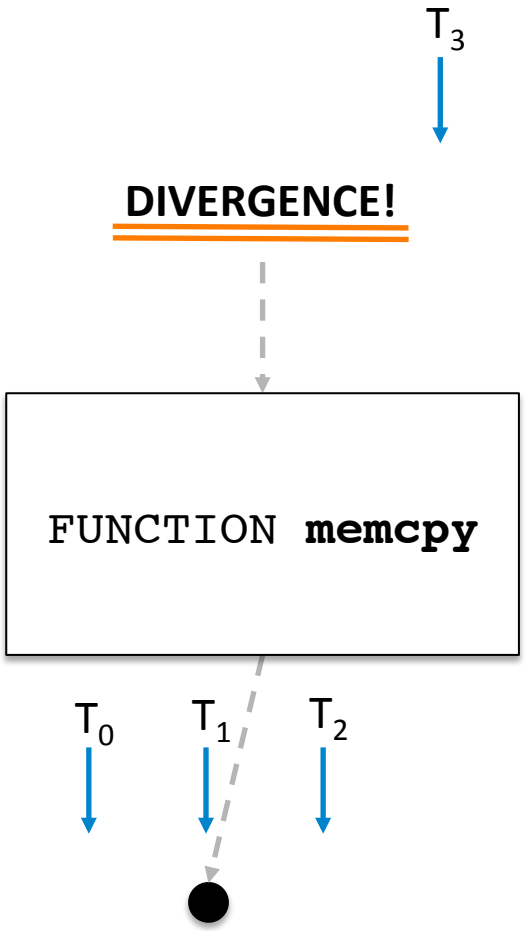
Control flow graph for memcpy.

Divergences: Coda

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

Kernel for parallel execution (CUDA).

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



Control flow graph for memcpy.

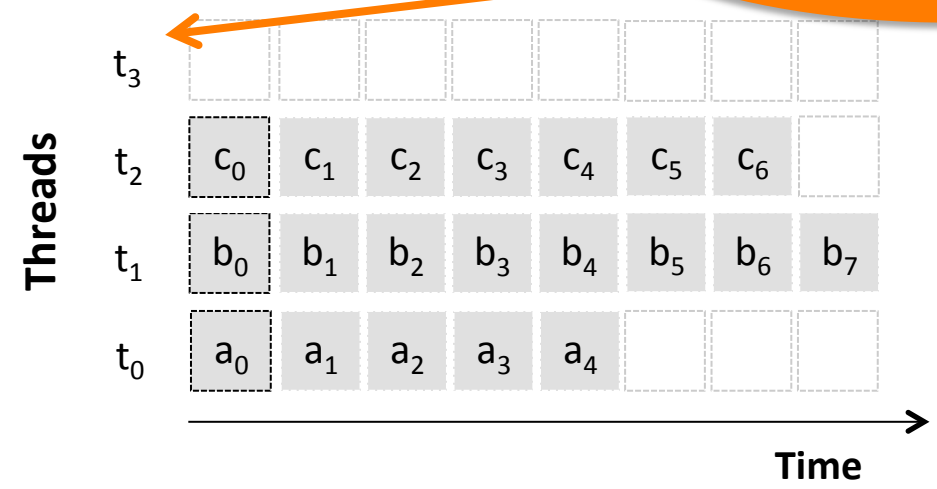
Divergences: Coda

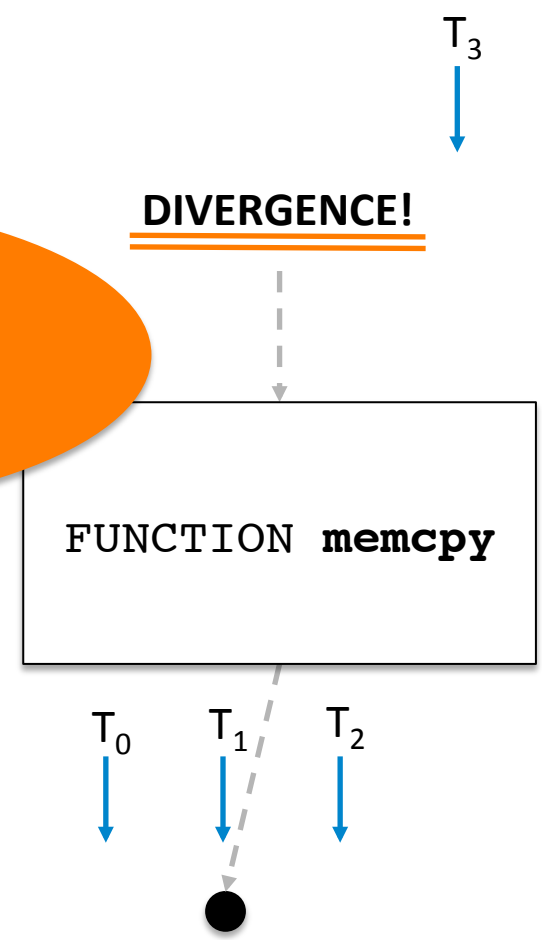
```
void kernel(int **A, int **B, int *N) {
  int tid(threadId.x);
  if (tid > N) {
    memcpy<<<1, 4>>>(A[tid], B[tid], N[tid]);
  } else {
    memcpy<<<1, 4>>>(B[tid], A[tid], N[tid]);
  }
}
```

Kernel for parallel execu

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

Observed behavior:





Control flow graph for memcpy.

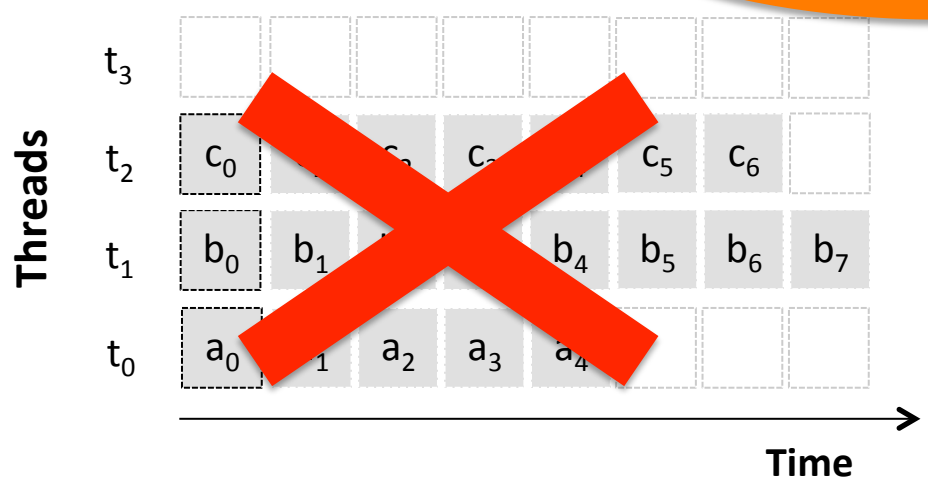
Divergences: Coda

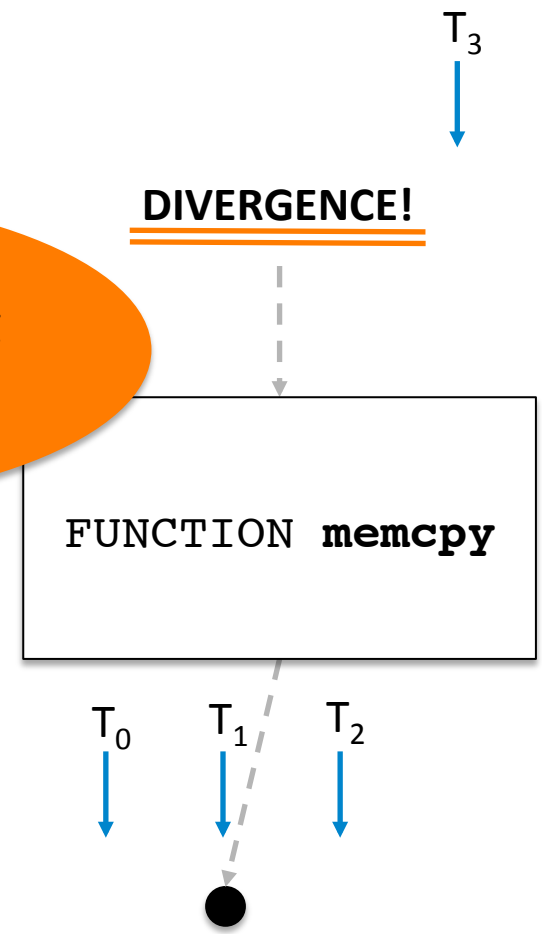
```
void kernel(int **A, int **B, int *N) {
  int tid(threadId.x);
  if (tid > N) {
    memcpy<<<1, 4>>>(A[tid], B[tid], N[tid]);
  } else {
    memcpy<<<1, 4>>>(B[tid], A[tid], N[tid]);
  }
}
```

Kernel for parallel execu

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

Observed behavior:





Control flow graph for memcpy.

Compilers Laboratory

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 > N) {
    memcpy<<<1, 4>>>(A[tid], B[tid], N[tid]);
  } else {
    memcpy<<<1, 4>>>(B[tid], A[tid], N[tid]);
  }
}
```

Kernel for parallel execution (CUDA).

CUDA's special syntax for **dynamic parallelism**:
kernel<<<#warps, #threads>>>(args…**)**

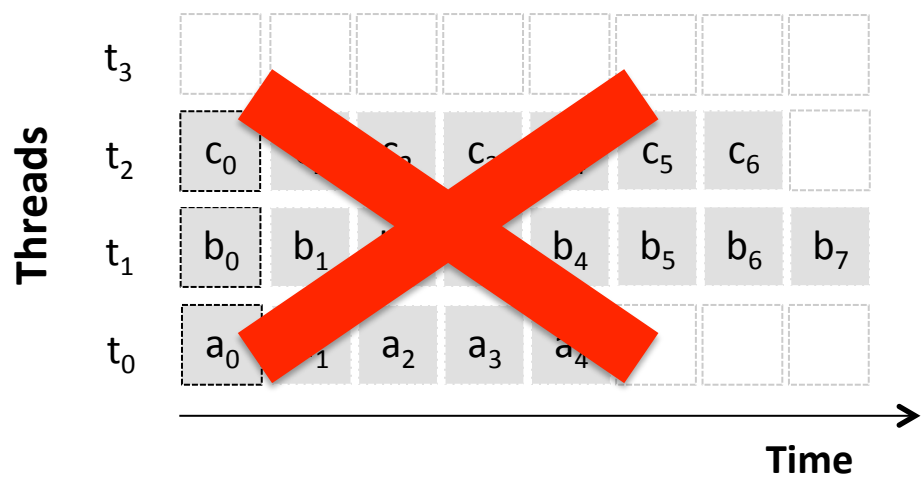
Dynamic Parallelism

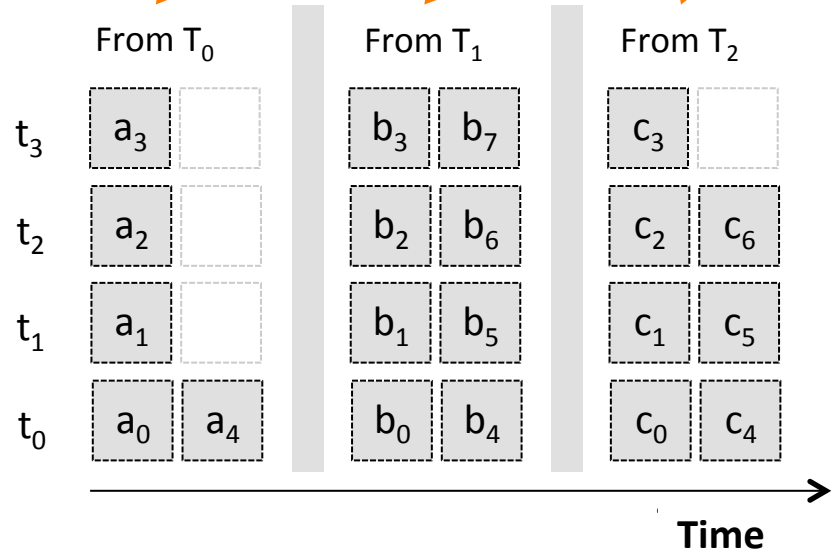
```
void kernel(int **A, int **B, int *N) {
  int tid(threadId.x);
  if (tid > N) {
    memcpy<<<1, 4>>>(A[tid], B[tid], N[tid]);
  } else {
    memcpy<<<1, 4>>>(B[tid], A[tid], N[tid]);
  }
}
```

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

Kernel for parallel execution (CUDA).

Actual behavior with CUDA's dynamic parallelism:





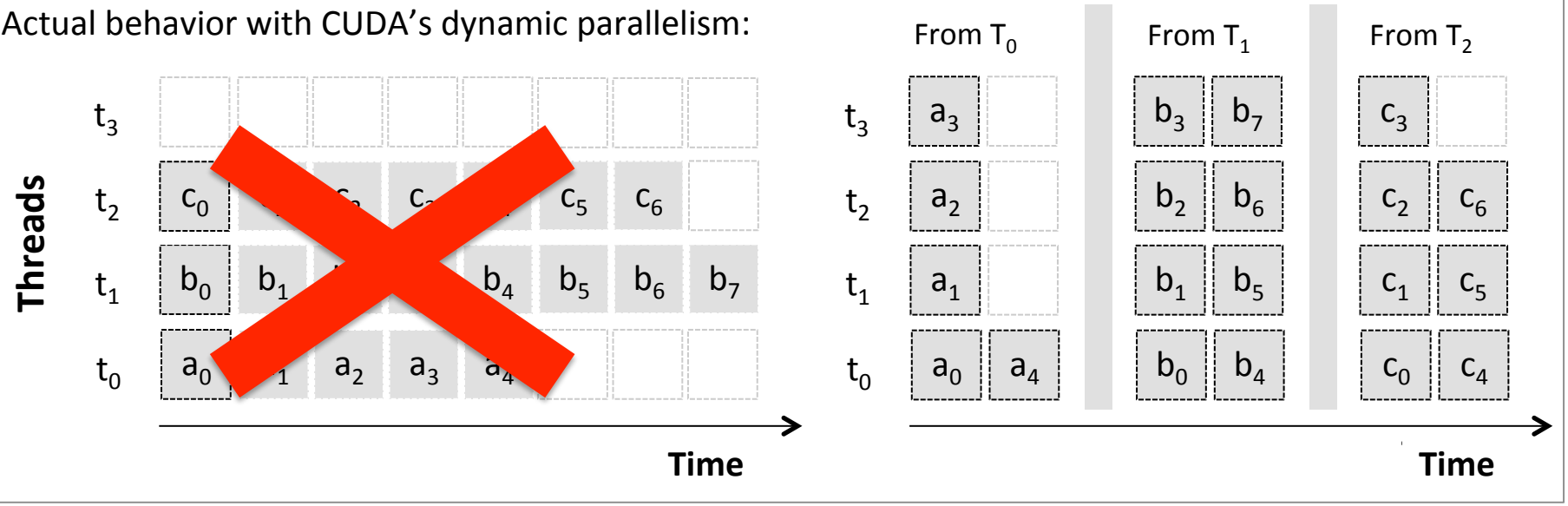
Dynamic Parallelism

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

SIMD kernels!

Kernel for parallel execution (CUDA).

Actual behavior with CUDA's dynamic parallelism:

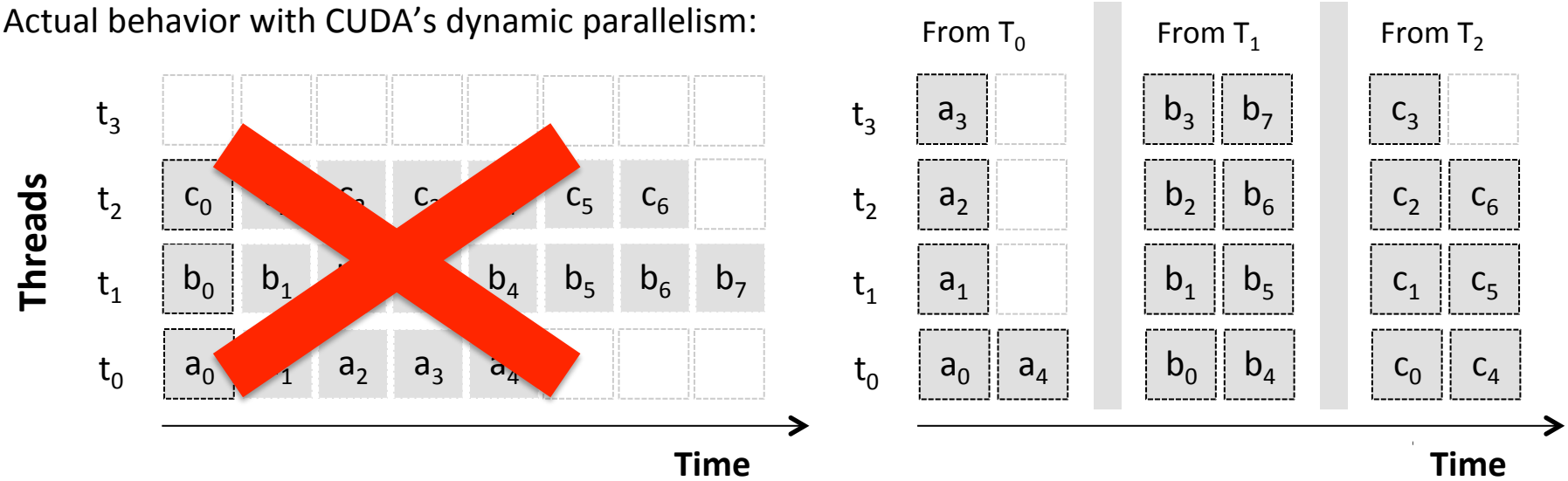


Dynamic Parallelism

```
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:



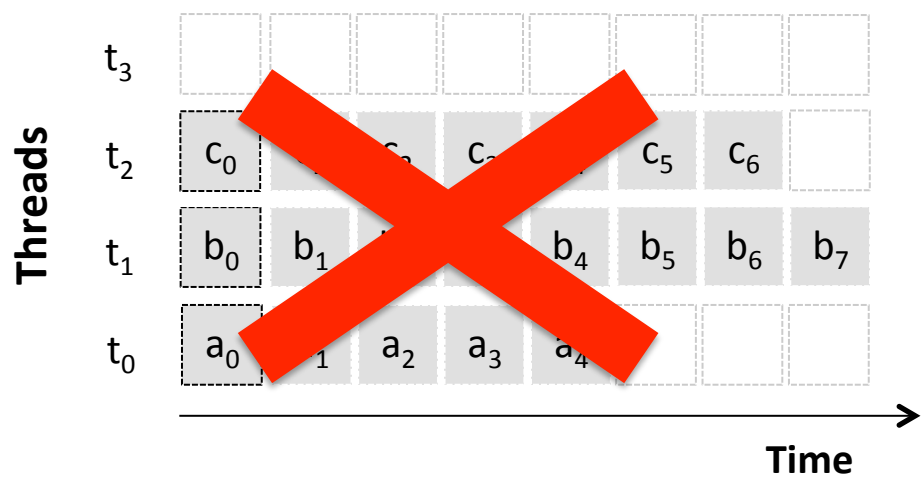
Dynamic Parallelism

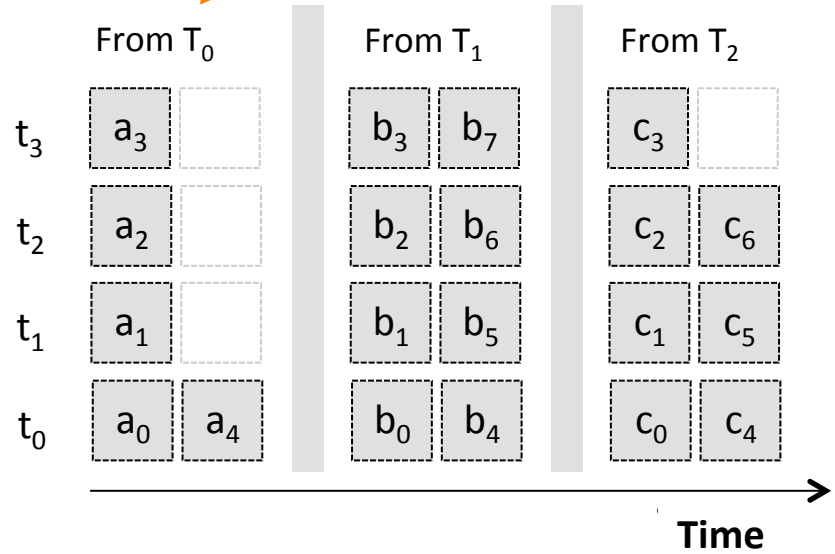
```
void memcpy(int *dest, int *src, int N) {
  for (int i=threadId.x; i < N; i+=threadDim.x) {
    dest[i] = src[i];
  }
}
```

SIMD implementation of memory copy.

All threads work on a single vector!

Actual behavior with CUDA's dynamic parallelism:





Dynamic Parallelism

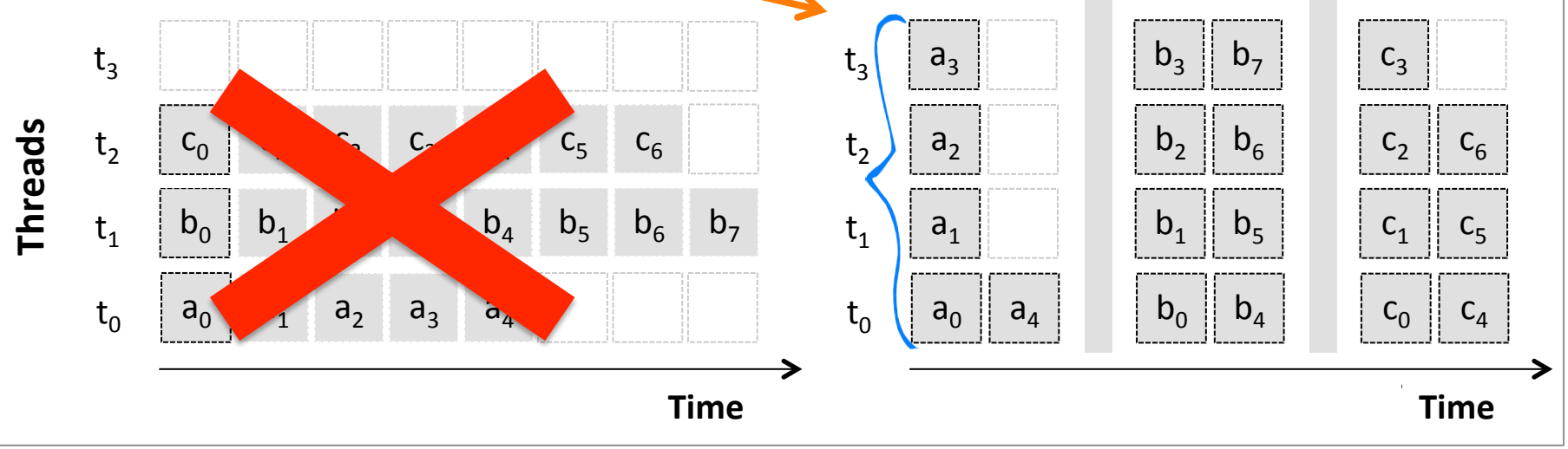
```
void memcpy(int *dest, int *src, int N) {
  for (int i=threadId.x; i < N; i+=threadDim.x) {
    dest[i] = src[i];
  }
}
```

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

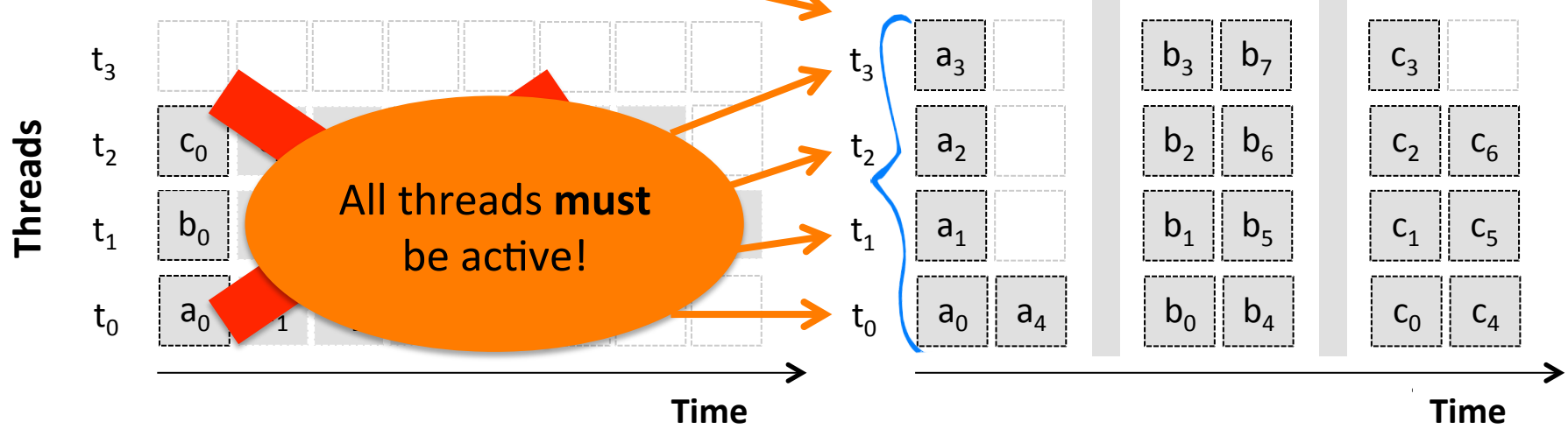
```
void memcpy(int *dest, int *src, int N) {
  for (int i=threadId.x; i < N; i+=threadDim.x) {
    dest[i] = src[i];
  }
}
```

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

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 > N) {
    memcpy<<<1, 4>>>(A[tid], B[tid], N[tid]);
  } else {
    memcpy<<<1, 4>>>(B[tid], A[tid], N[tid]);
  }
}
```

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

Kernel for parallel execution (CUDA).

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 > N) {
    memcpy<<<1, 4>>>(A[tid], B[tid], N[tid]);
  } else {
    memcpy<<<1, 4>>>(B[tid], A[tid], N[tid]);
  }
}
```

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 > N) {
    memcpy<<<1, 4>>>(A[tid], B[tid], N[tid]);
  } else {
    memcpy<<<1, 4>>>(B[tid], A[tid], N[tid]);
  }
}
```

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 + $\frac{\text{Sequential Time}}{\#\text{warps} \times \#\text{threads}}$

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 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.

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.

Mappings:

T_0 T_1 T_2 T_3 $\qquad\qquad$ T_0 T_1 T_2 T_3

int value = [10 20 30 10] $\xrightarrow{\text{increment}}$ **int** value = [11 21 31 11]

Reductions:

T_0 T_1 T_2 T_3 $\qquad\qquad$ T_0 T_1 T_2 T_3

int value = [10 20 30 10] $\xrightarrow{\text{sum}}$ **int** scalar = (70 70 70 70)

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:

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

```
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

```
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

```
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 mem

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

```
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

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

SIMD implementation c

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

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

SIMD implementation c

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
    }
  }
}
```

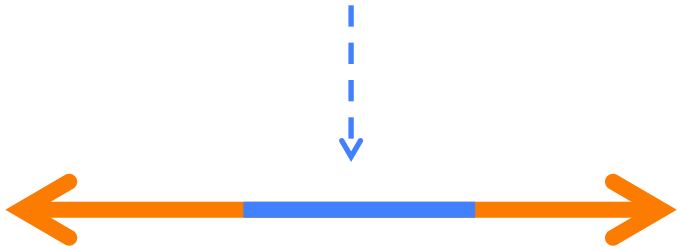
NVIDIA hardware **does not** support re-enabling of threads within warp!

Warp-synchronous wrapper for SIMD memory copy.

Warp-Synchronous Programming: Everywhere blocks

`everywhere`

temporarily re-enables all threads within the warp



SPMD

handle divergences

SIMD

all threads must be active
at the call site

Warp-Synchronous Programming: Everywhere blocks

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Warp-Synchronous Programming: Everywhere blocks

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.

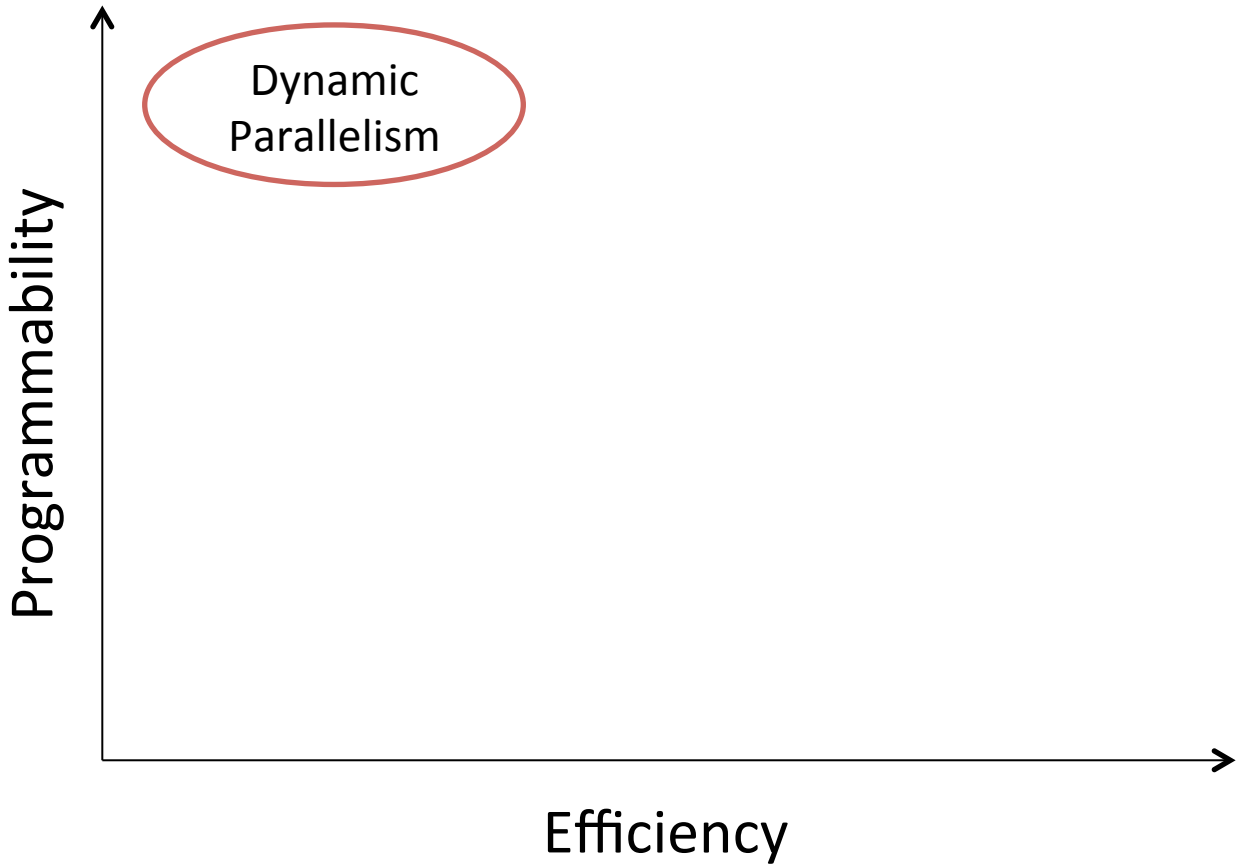
Compilers Laboratory

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

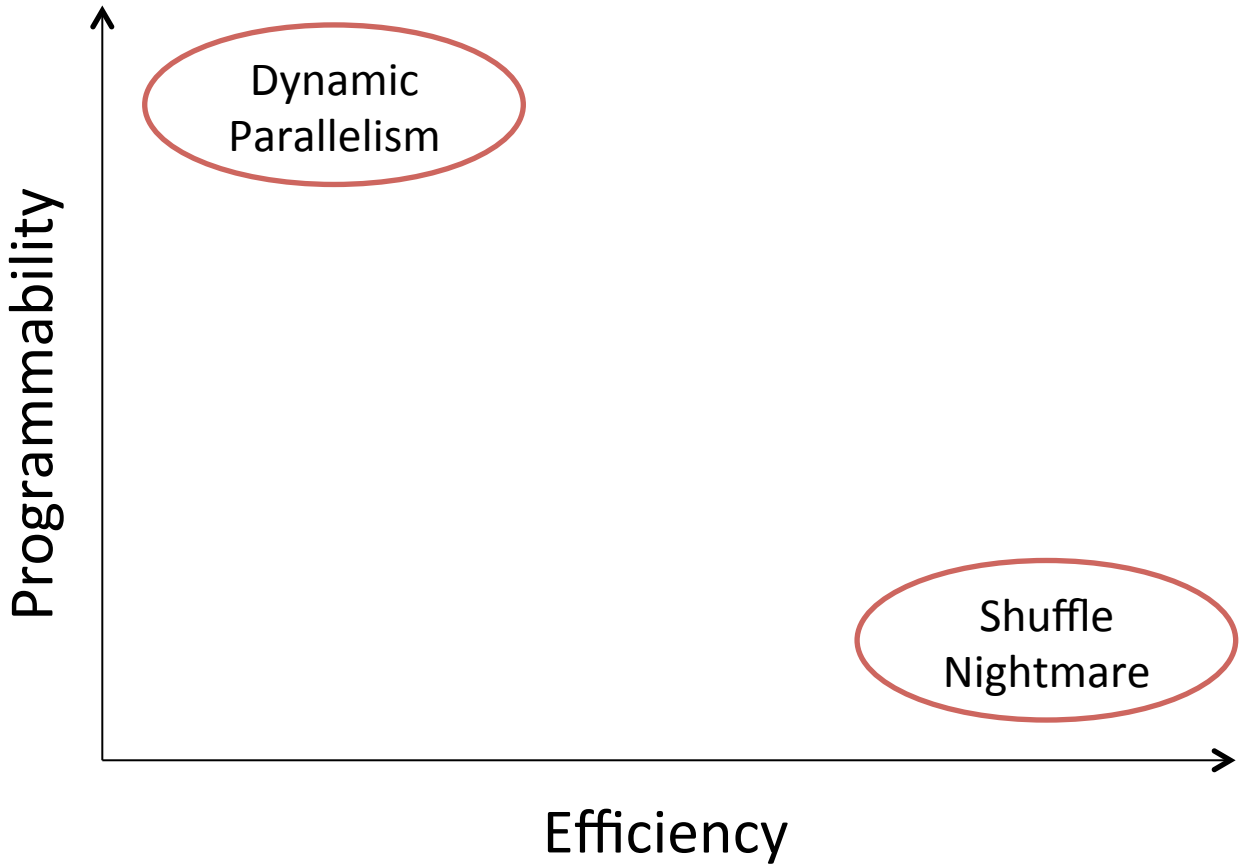
Function Call Re-vectorization

Function Call Re-Vectorization

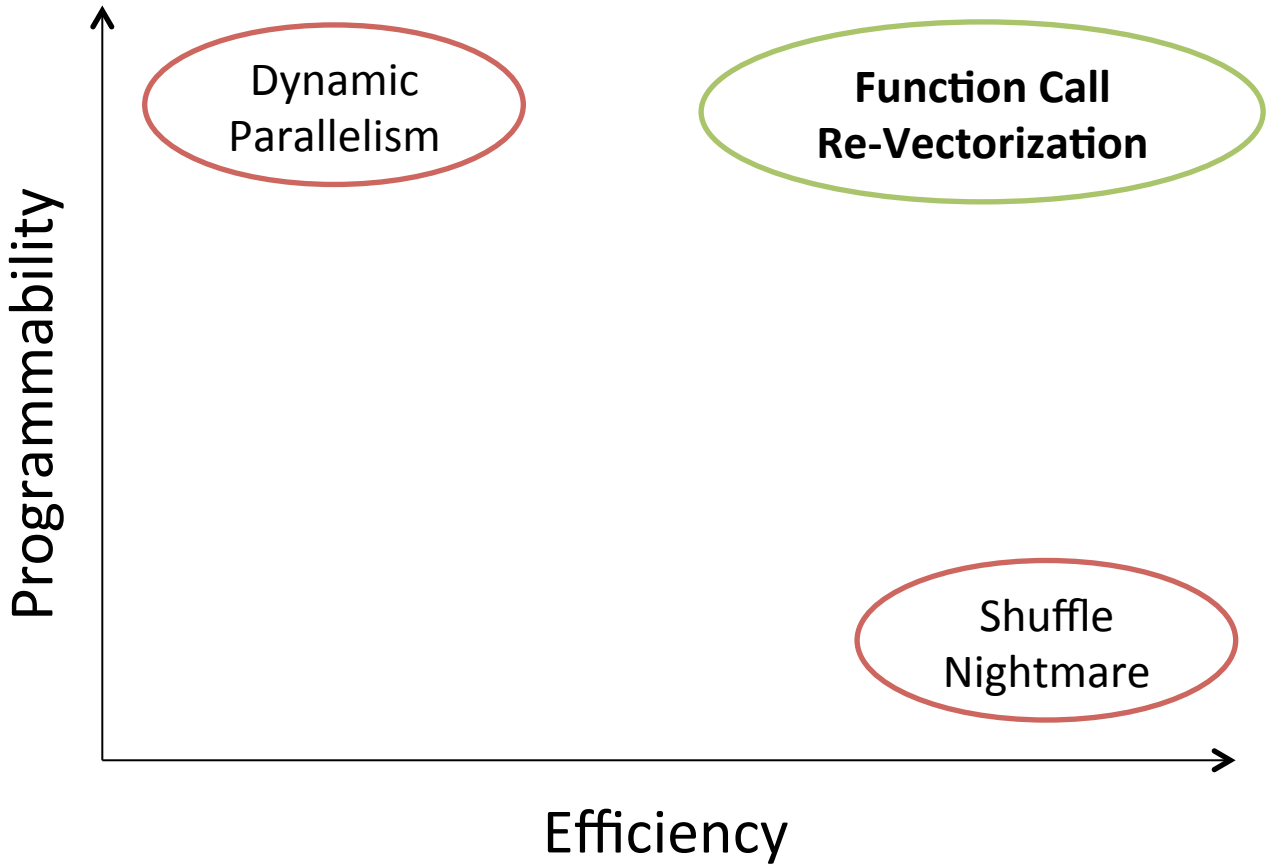
Function Call Re-Vectorization



Function Call Re-Vectorization

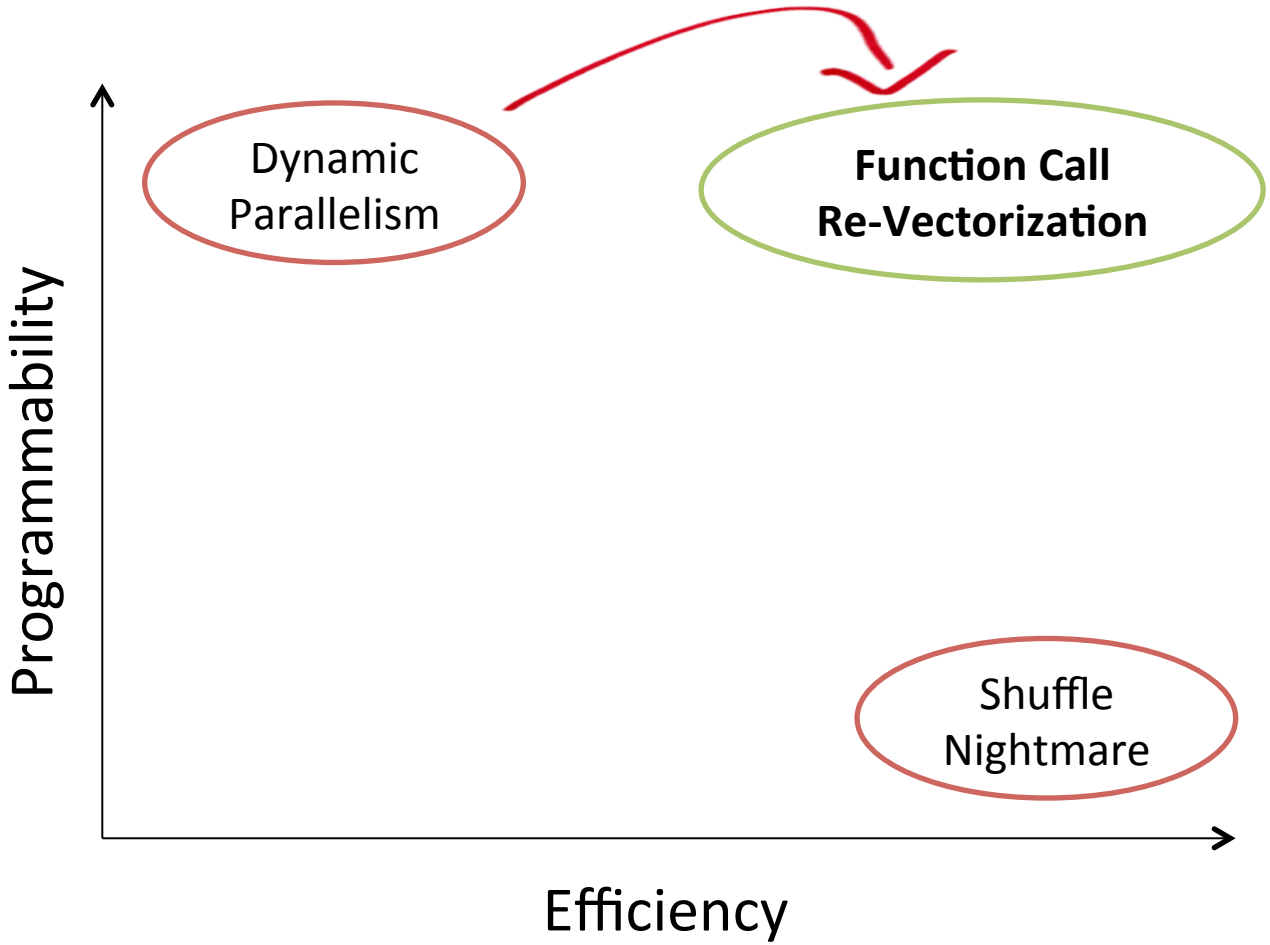


Function Call Re-Vectorization

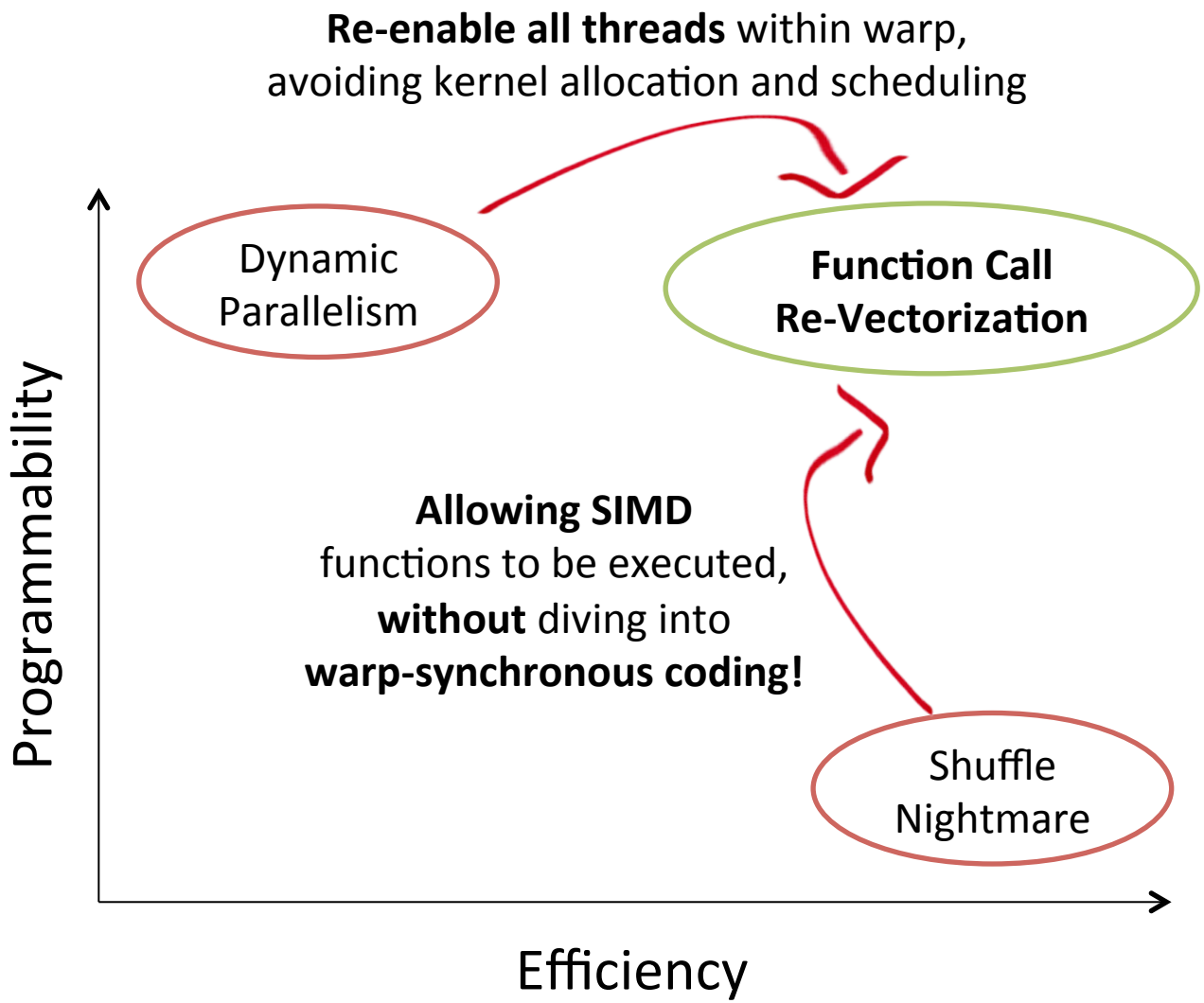


Function Call Re-Vectorization

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



Function Call Re-Vectorization



Function Call Re-Vectorization: CREV

CREV

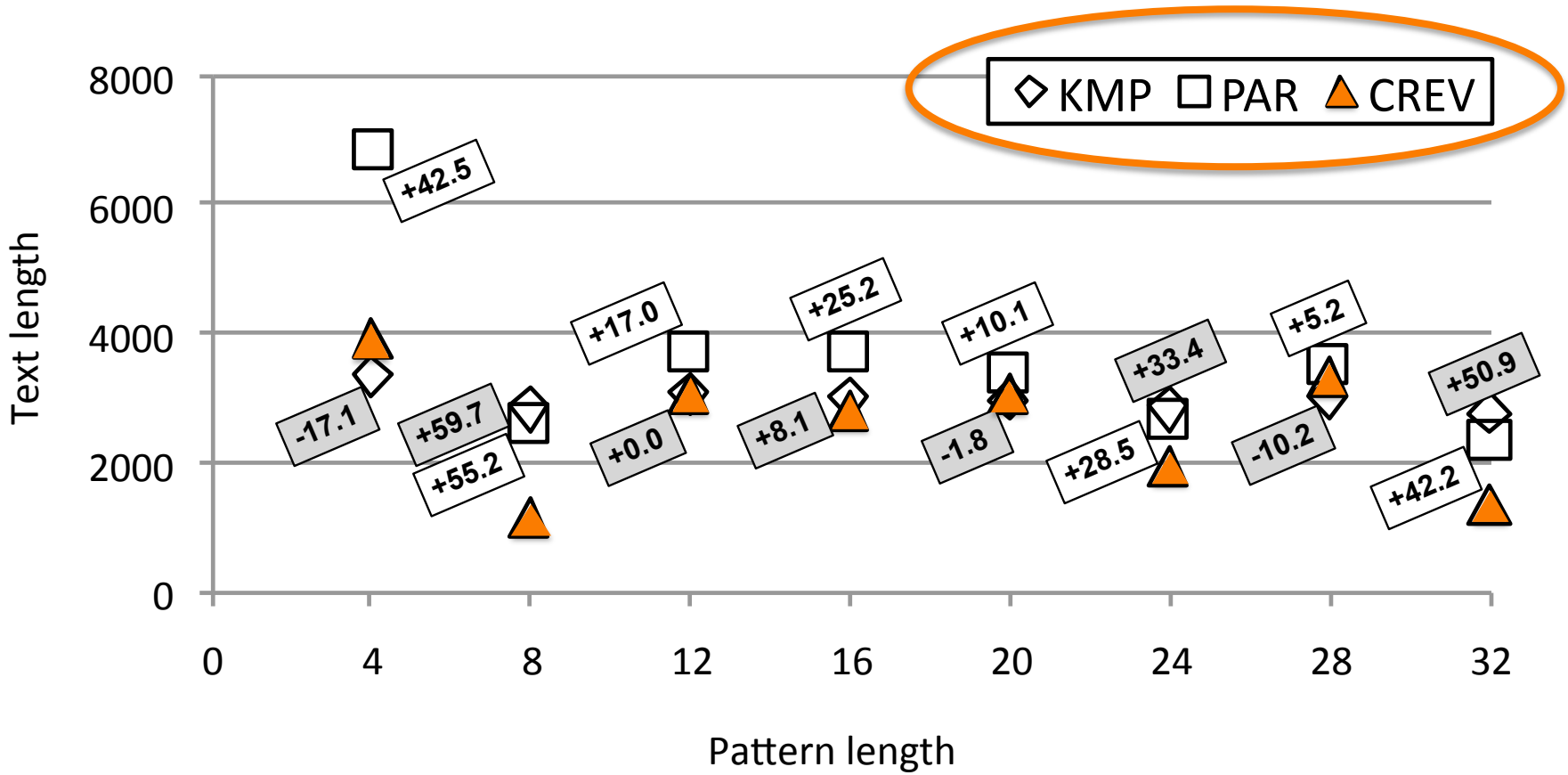
```
string T = text, P = pattern;

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

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

SPMD function with **crev** call within **divergent region**

Naïve parallel approach

Function Call Re-Vectorization: CREV



Function Call Re-Vectorization: CREV

Properties of CREV:

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

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

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

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

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	-- not implemented	529856.065	*erdos-renyi*
Depth-First Search	3754.101	3786.263	-- not implemented	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).

Compilers Laboratory

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

Questions?

Sergey Prokofiev – Piano sonata no. 7 op. 83

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

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