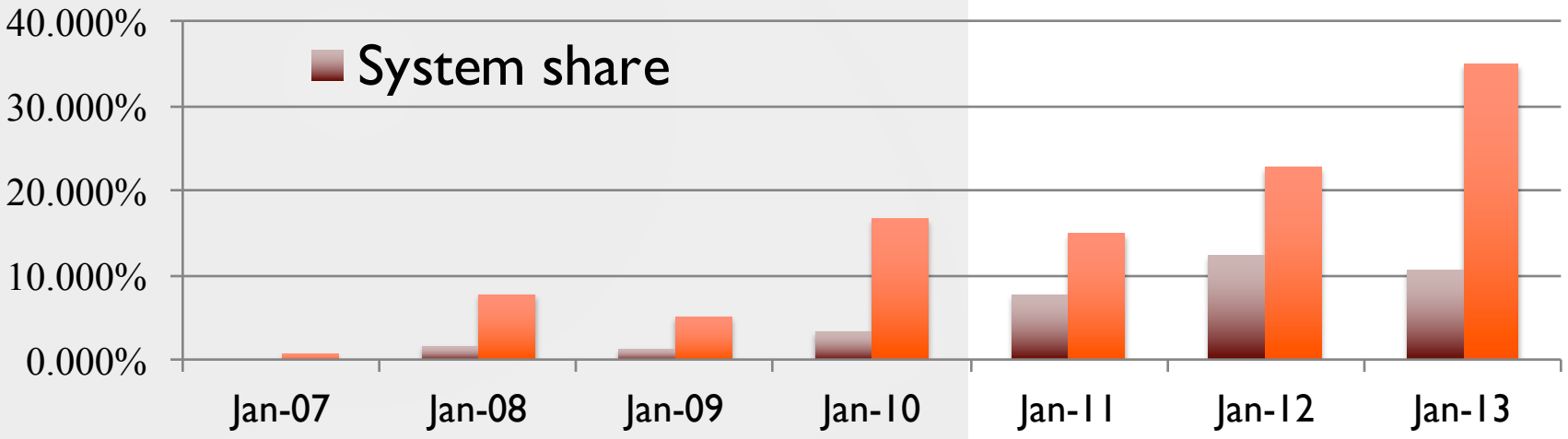
CoreTSAR: Adaptive Worksharing for Heterogeneous Systems

Tom Scogland, Wu-chun Feng
Dept. of Computer Science, Virginia Tech
Barry Rountree, Bronis R. de Supinski
Lawrence Livermore National Laboratory

Edited and presented by Harsh Khetawat

Heterogeneity in HPC: Accelerators on the Top500



- Heterogeneity becoming the norm in HPC
- Top 10 HPC systems dominated by accelerated systems

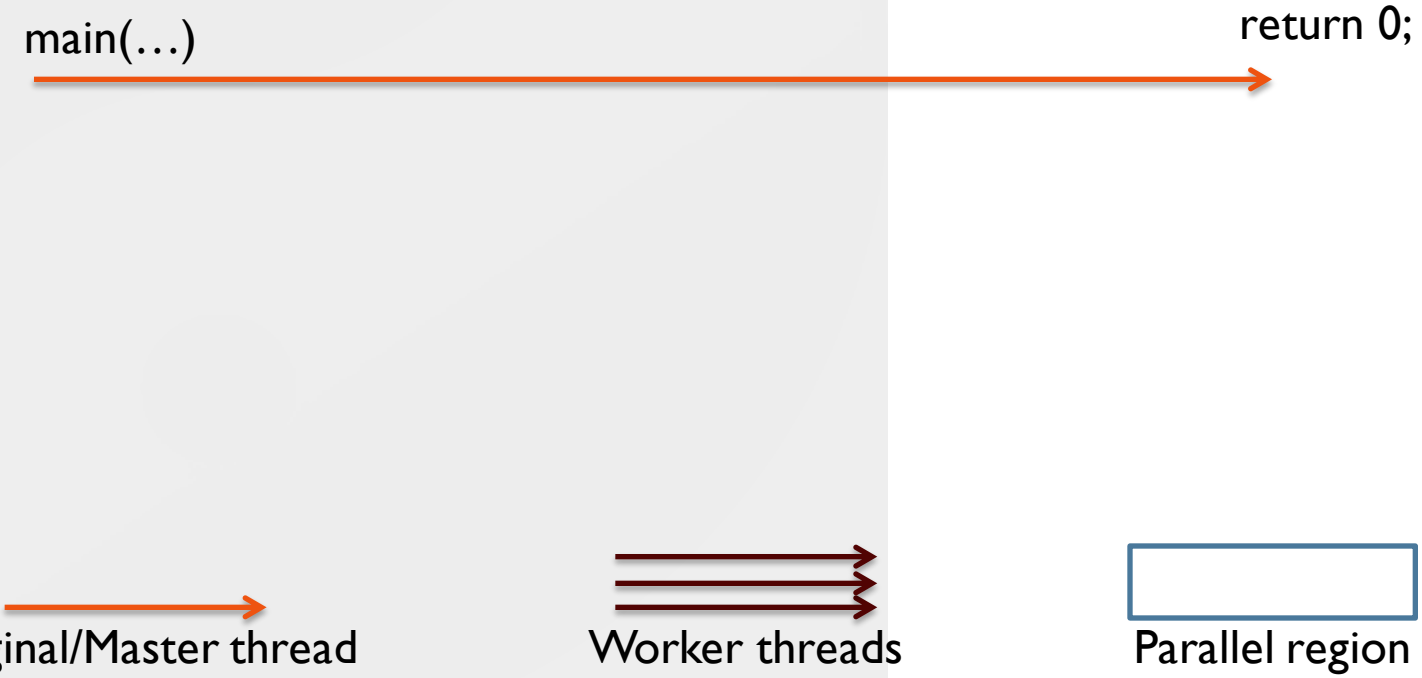
How do we target them?

General Matrix Multiplication(GEMM): CPU serial

```
void runGemm(T *a, T *b, T *c) {
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; ++j) {
      c[(i * N) + j] *= B;
      for (int k = 0; k < N; ++k) {
        c[(i*N) + j] += A * a[(i*N) + k] * b[(k*N) + j];
      }
    }
  }
}
```

Targets: One CPU core

Threading Behavior

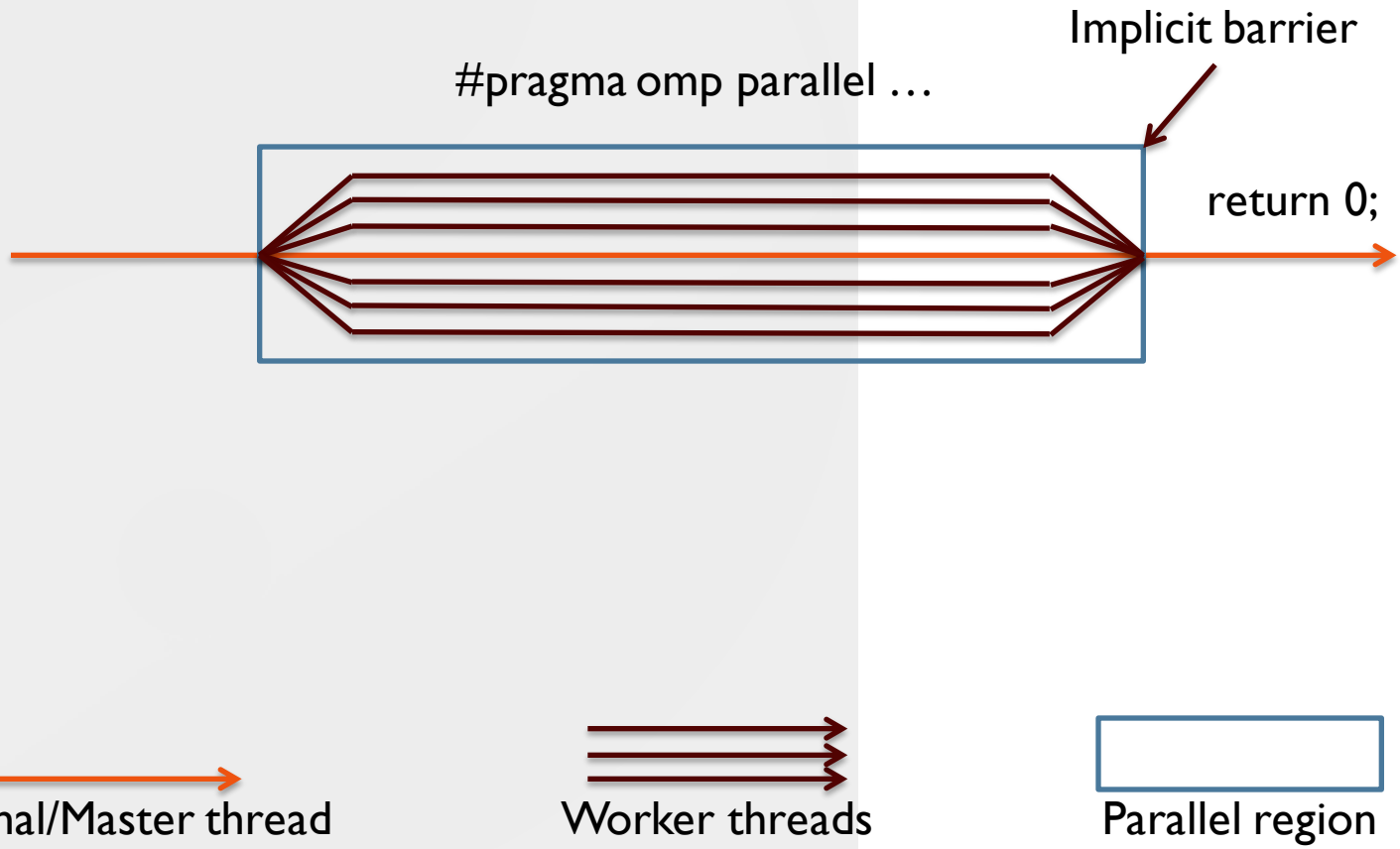


GEMM: OpenMP

```
void runGemm(T *a, T *b, T *c) {
#pragma omp parallel for
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; ++j) {
      c[(i * N) + j] *= B;
      for (int k = 0; k < N; ++k) {
        c[(i*N) + j] += A * a[(i*N) + k] * b[(k*N) + j];
      }
    }
  }
}
```

Targets: All local CPU cores

OpenMP Basic Behavior



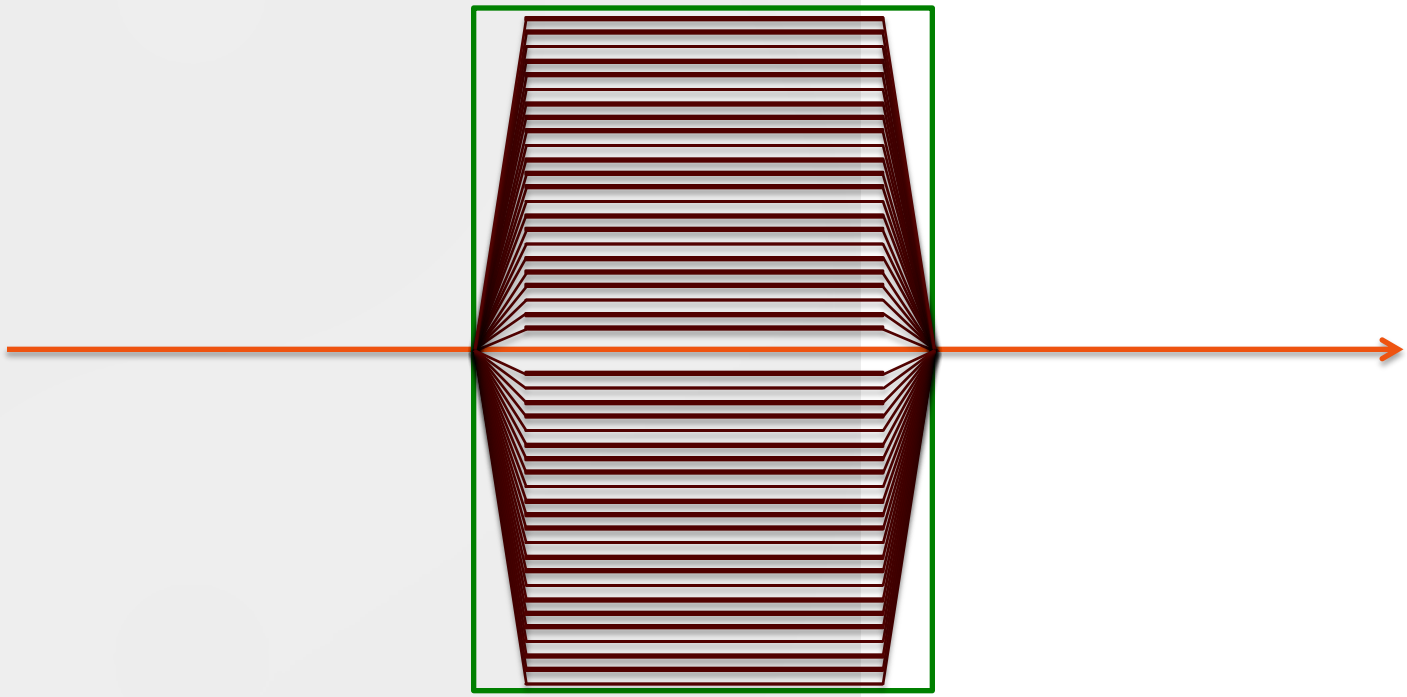
GEMM: CUDA (minimal)

```
__global__ void cudag(T *a, T *b, T *c, TA, TB, int n) {
 uint i = blockIdx.x * blockDim.x + threadIdx.x;
 if(i < n) {
  for (int j = 0; j < n; ++j) {
   c[(i*N) + j] *= B;
   for (int k = 0; k < n; ++k) {
    c[(i*N)+j] += A * a[(i*N)+k] * b[(k*N)+j];
} } } }
void runGemm(T **a, T **b, T **c) {
 T *ca, *cb, *cc; dim3 dB, dG;
 size_t size = N*N*sizeof(T);
 dB.x = 64; dB.y = dB.z = 1;
 dG.x = (N/dB.x)+1; dG.y = dG.z = 1;
 cudaMalloc(&ca, size);
 cudaMalloc(&cb, size);
 cudaMalloc(&cc, size);
 cudaMemcpy(ca,*a,size,cudaMemcpyHostToDevice);
 cudaMemcpy(cb,*b,size,cudaMemcpyHostToDevice);
 cudaMemcpy(cc,*c,size,cudaMemcpyHostToDevice);
 cudag<<<dG,dB>>>(a, b, c, A, B, N);
 cudaMemcpy(*c,cc,size,cudaMemcpyDeviceToHost);
}
```

Targets: **One** GPU

CUDA Threading Behavior

cudag<<<dG,dB>>>(a, b, c, A, B, N);



Original/Master thread

Worker threads

Parallel region

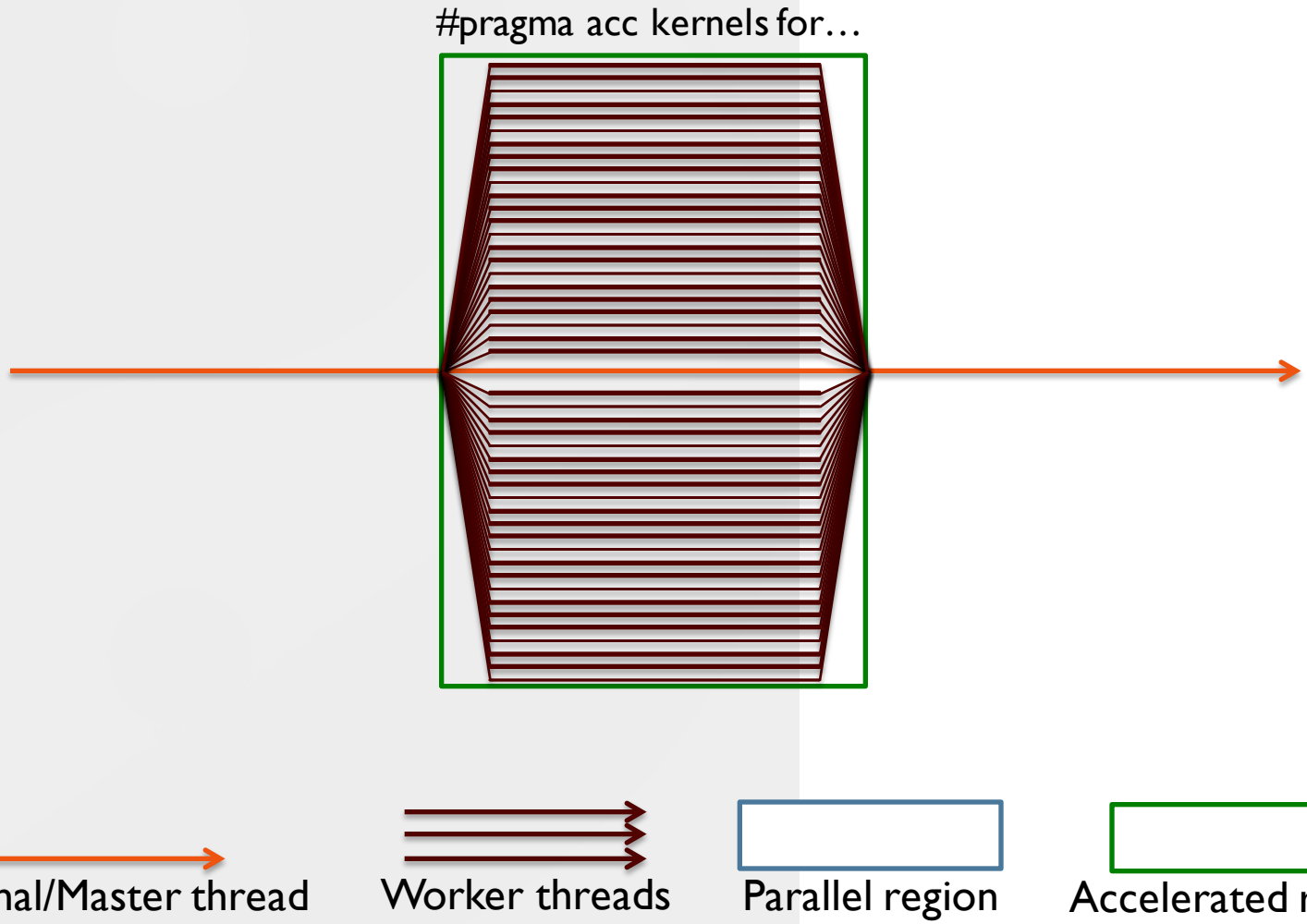
Accelerated region

GEMM: Accelerated OpenMP (OpenMP 4.0 syntax)

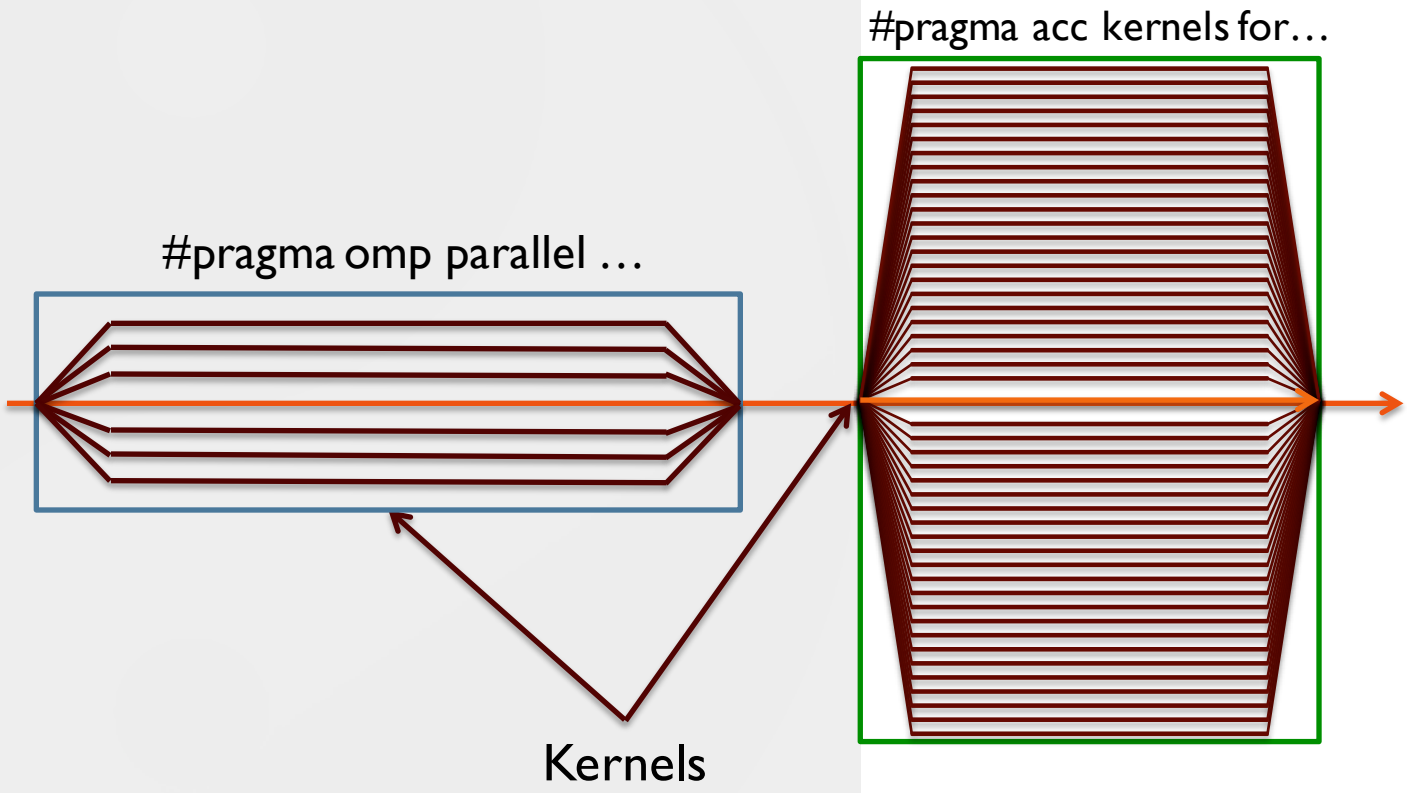
```
void runGemm(T *a, T *b, T *c) {
#pragma omp target teams distribute parallel for \
                map(tofrom: c[0:N][0:N])   \
        map(to: a[0:N][0:N], b[0:N][0:N])
for (int i = 0; i < N; i++) {
   for (int j = 0; j < N; ++j) {
     c[(i*N) + j] *= B;
     for (int k = 0; k < N; ++k) {
       c[(i*N) + j] += A * a[(i*N) + k] * b[(k*N) + j];
     }
   }
  }
}
```

Targets: **One** accelerator or **one** CPU core

Accelerated OpenMP Threading Behavior



Accelerated OpenMP + OpenMP Threading Behavior



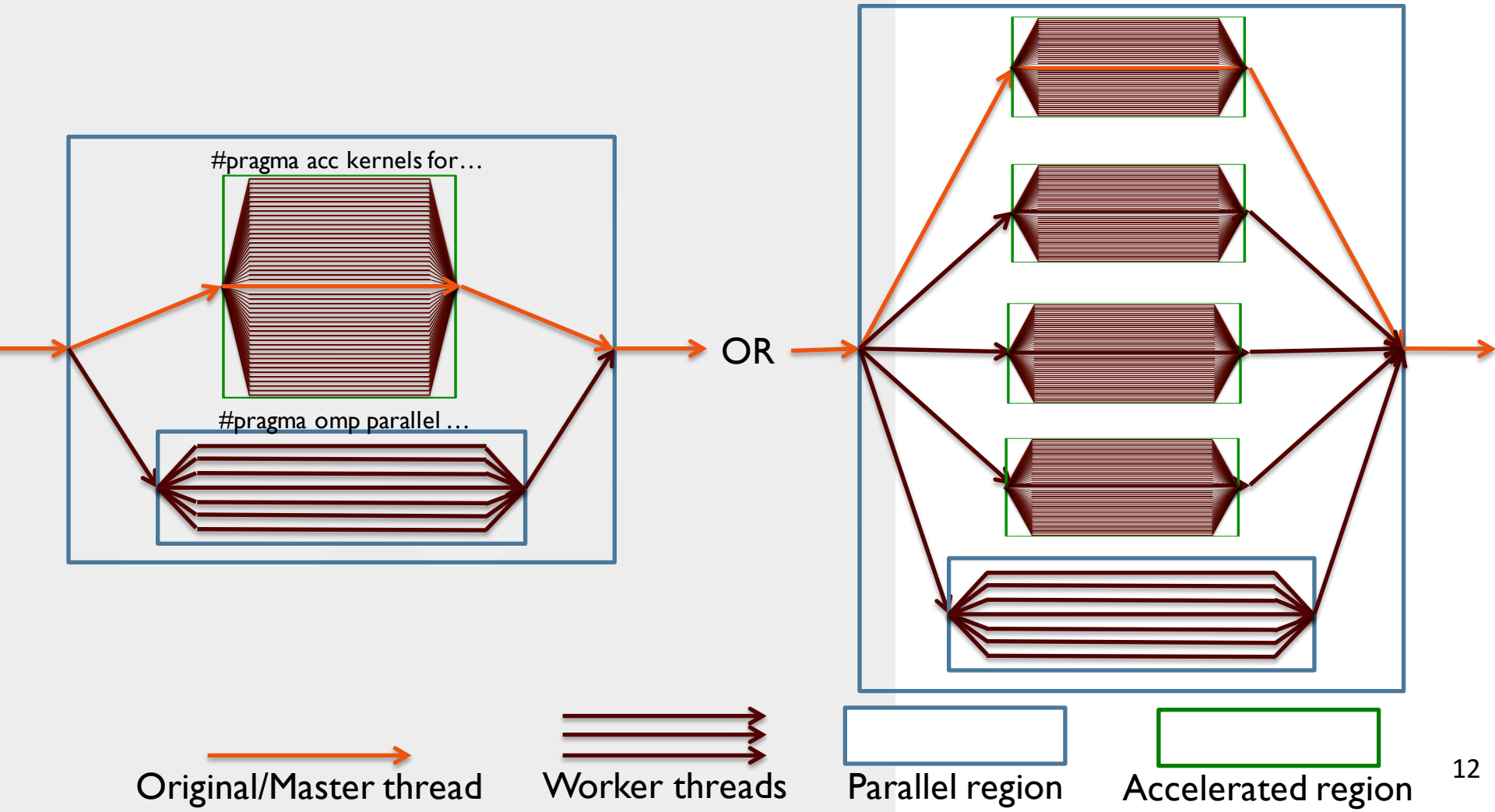
Original/Master thread

Worker threads

Parallel region

Accelerated region

The Goal:
Work-share a Target Region Across the Whole System



Issues

- Computational Overhead
  - Launching tasks onto GPUs is **expensive** (compared to CPUs)
- Heterogeneity of Computational Devices
  - Mapping the right amount of work to the right core
- Memory Incoherence
  - CPUs and GPUs generally do ***not*** share memory

Solutions

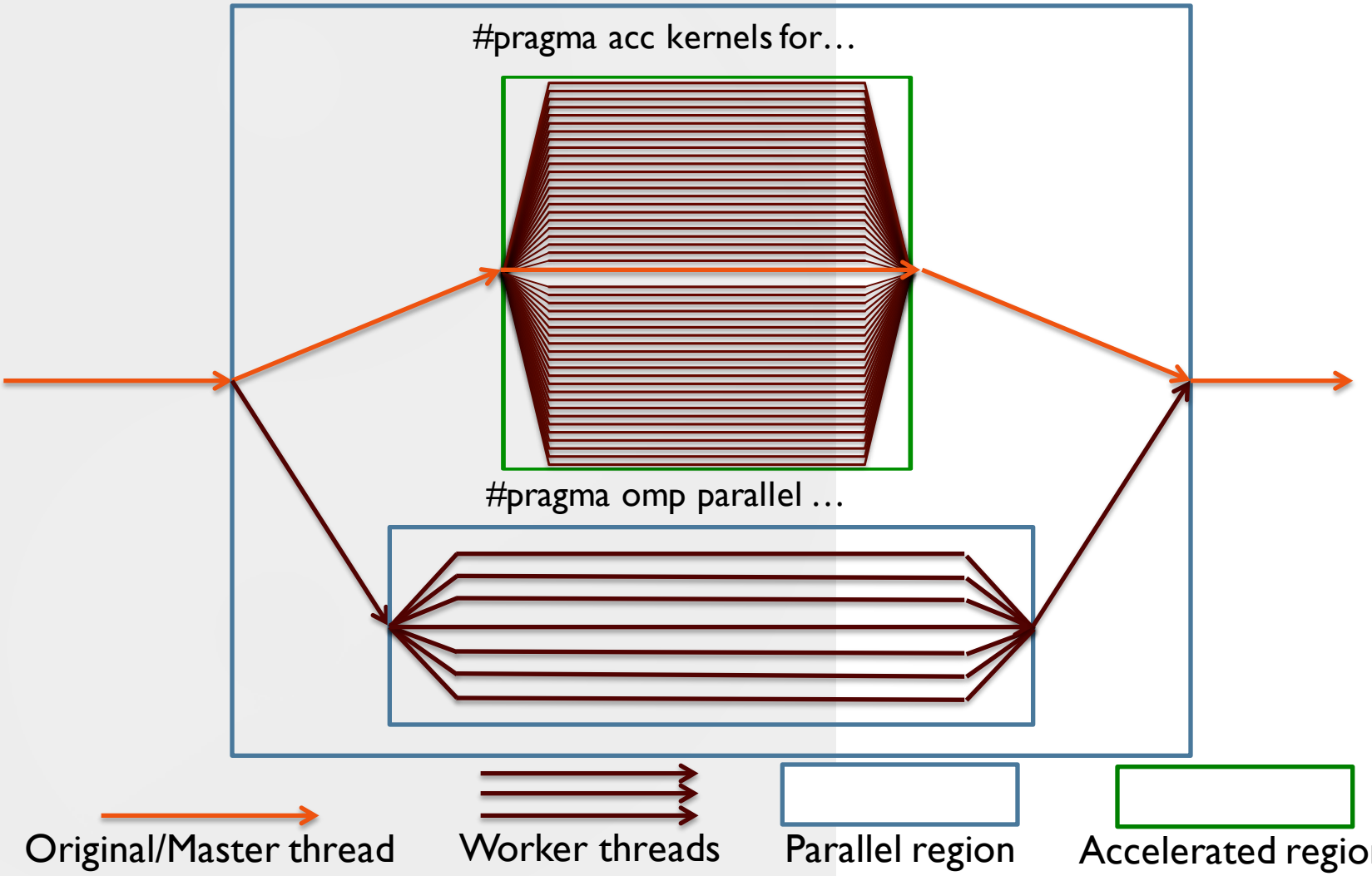
- Adaptive scheduling to minimize launch overhead (Splitter)
- Performance-prediction for load-balancing (CoreTSAR)
- Multi-device memory management (CoreTSAR)

Background:
GEMM: Our Previous Approach, Splitter

```
void runGemm(T *a, T *b, T *c) {
  splitter * s = split_init(N, SPLIT_DYNAMIC, NULL, NULL);
  for (int d_it = 0; d_it < s->d_end; d_it++) {
    s = split_next(no, d_it);
#pragma omp parallel num_threads(2)
    {
      int tid = omp_get_thread_num();
      if(tid == 0) split_gpu_start(s); else split_cpu_start(s);
      int start = tid == 0 ? s->gts : s->cts;
      int end = tid == 0 ? s->gts : s->cts;
#pragma omp target teams distribute parallel for          \
      map(tofrom: c[start:end][start:end])  \
      map(to: a[start:end][start:end])\
      map(to: b[start:end][start:end])
      for (int i = start; i < end; i++) {
        for (int j = 0; j < N; ++j) {
          c[(i*N)+j] *= B;
          for (int k = 0; k < N; ++k) {
            c[(i*N)+j] += A * a[(i*N)+k] * b[(k*N)+j];
      } } }
      if(tid == 0) split_gpu_end(s); else split_cpu_end(s);
} } }
```

Targets: **One** local accelerator **and** all CPU cores

Background: Splitter Threading Behavior

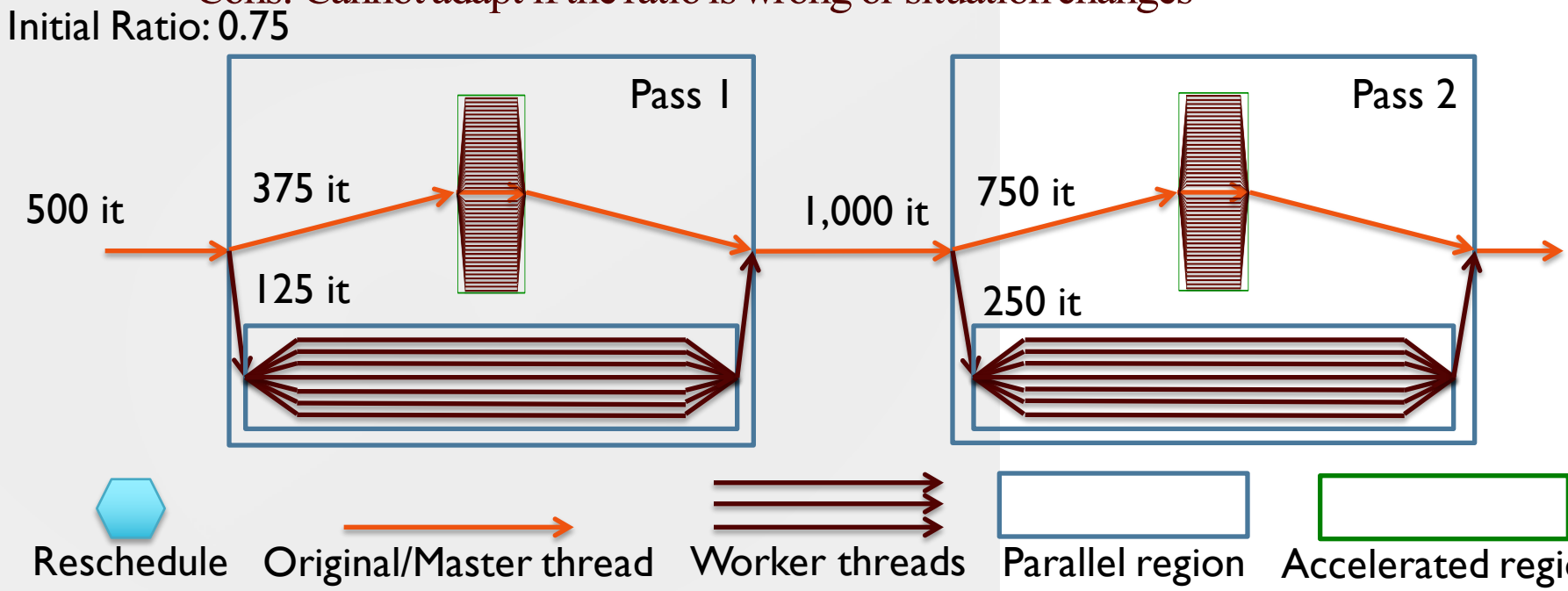


Background: Splitter's Main Contribution: Adaptive Static Scheduling Policies

- Minimize blocking time
  - No device should stand idle waiting for others
- Avoid overhead of running more tasks than necessary
  - Launching a task on a GPU, and moving memory, is expensive

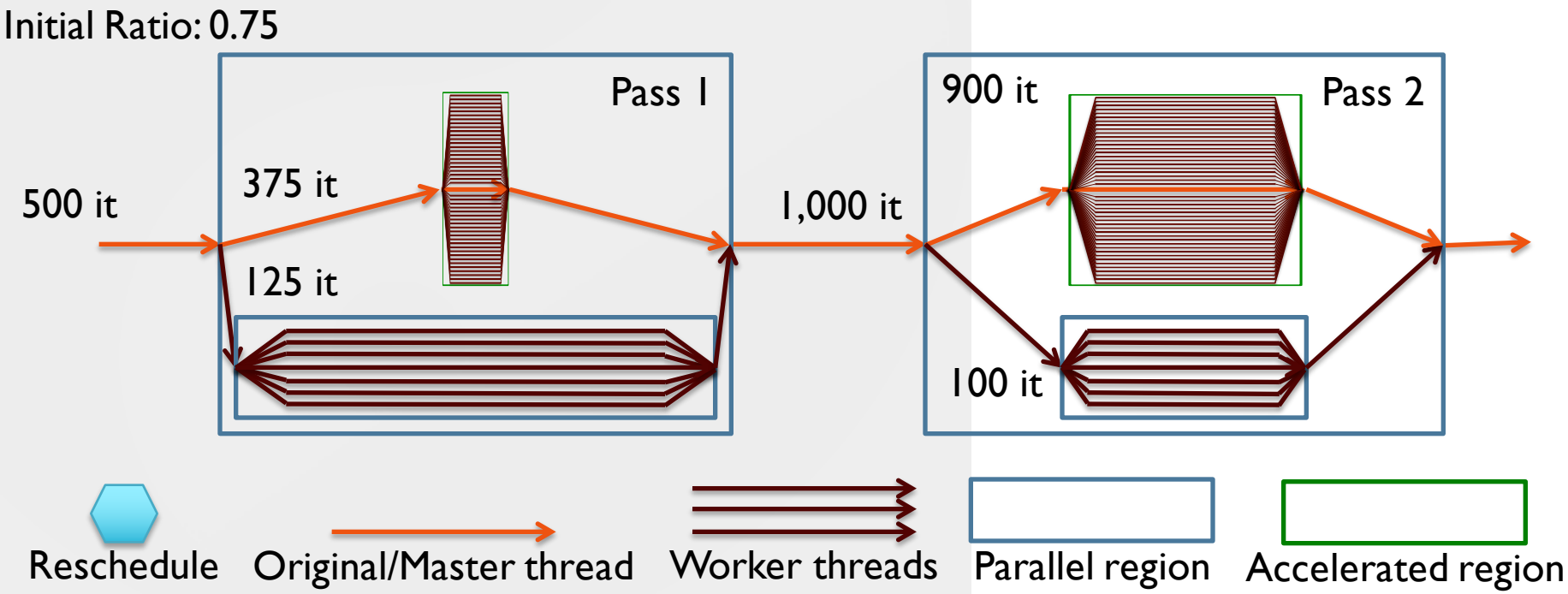
Background: Scheduling Policies: Static

- Divide work based on a static ratio
  - Straightforward extension of OpenMP static to divide work unevenly among targets, based on floating point performance by default
  - Pros: No scheduling overhead
  - Cons: Cannot adapt if the ratio is wrong or situation changes



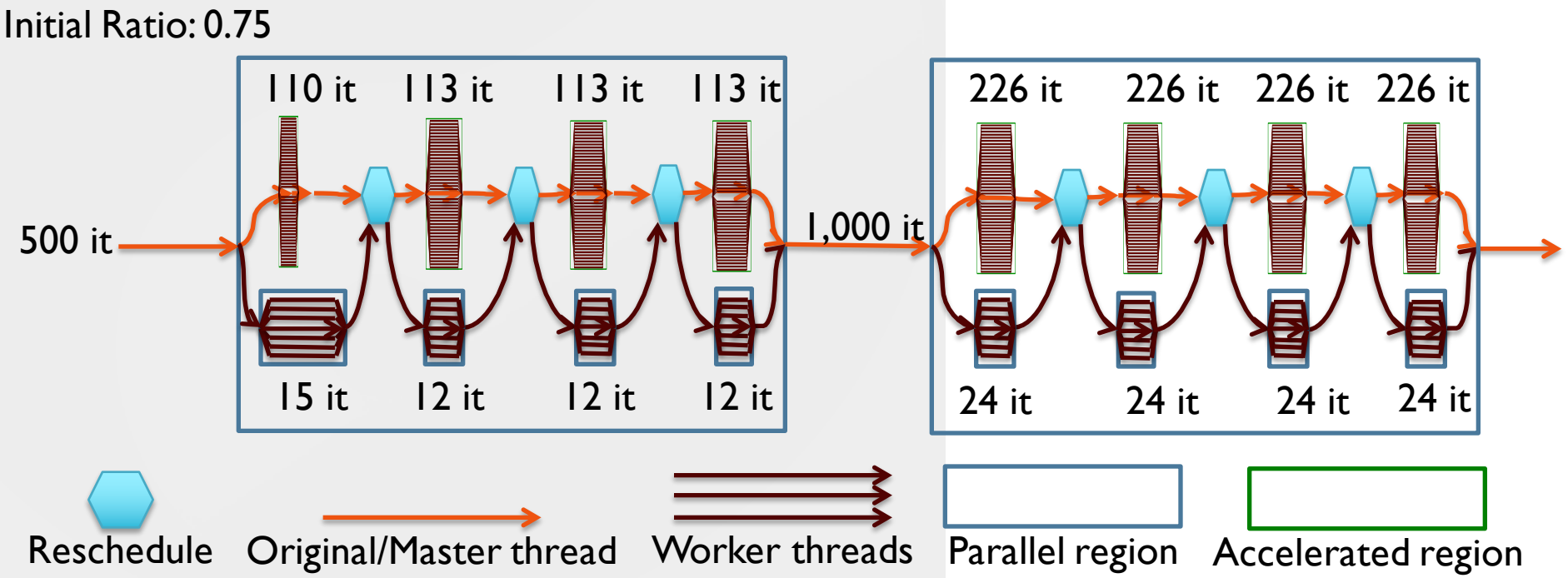
Background: Scheduling Policies: Adaptive

- Generate a ratio based on behavior each pass
  - Predicts the runtime on each device and minimizes blocking time
  - Pros: Creates one task per device, can adapt to change
  - Cons: Only adapts at entry to the region



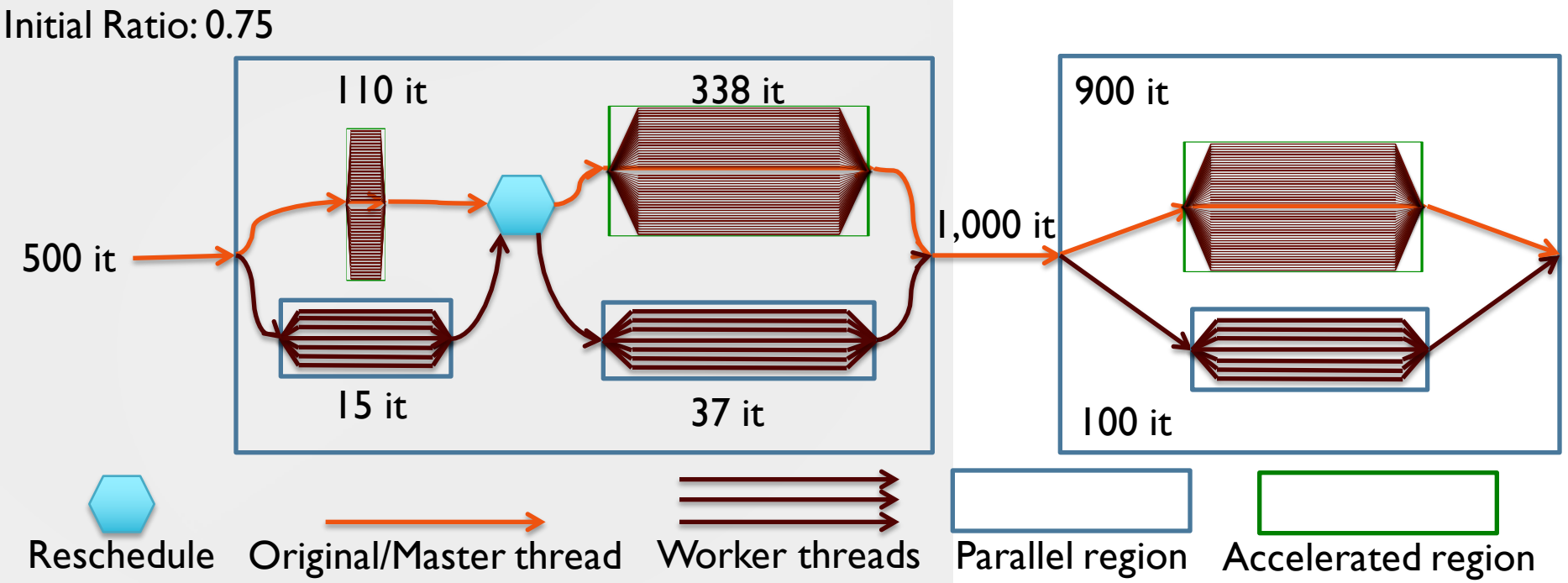
Background: Scheduling Policies: Split

- Breaks each pass into sub-passes and adapts for each
  - Takes an extra parameter, "div," for number of sub-passes to create
  - Pros: Adapts faster and more often
  - Cons: Higher overhead



Background: Scheduling Policies: Quick

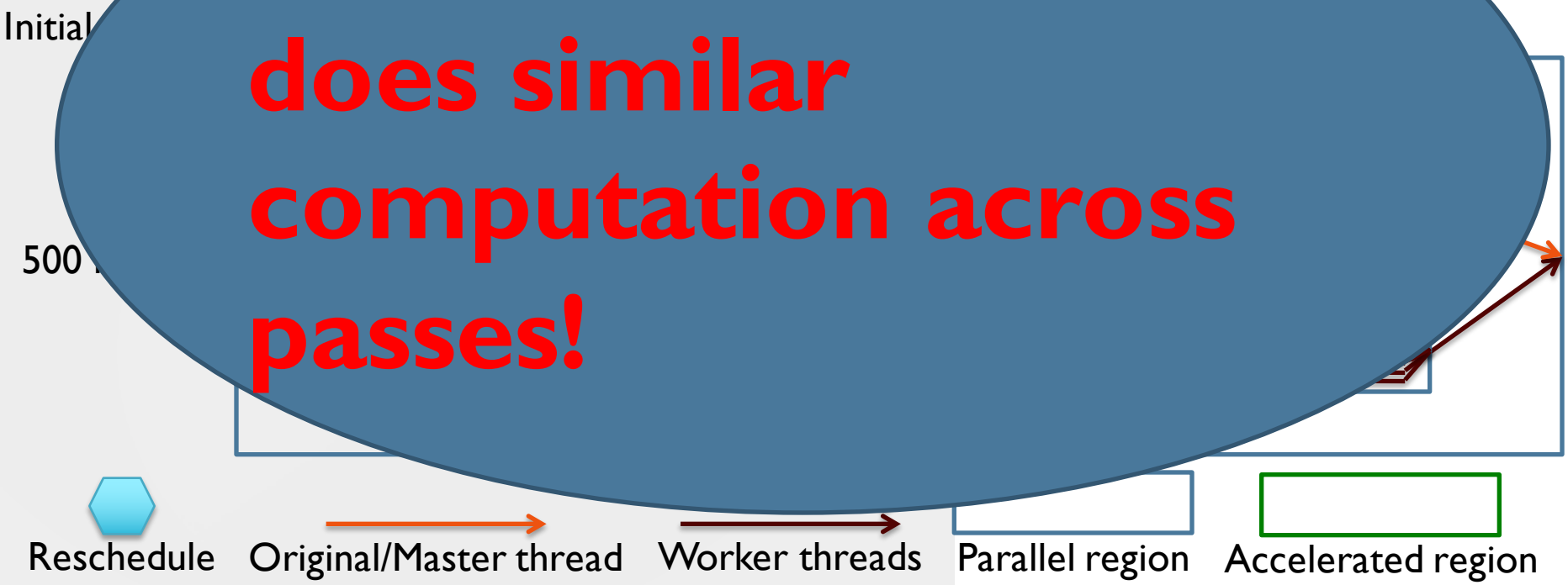
- Breaks the **first** pass into sub-passes to train
  - Uses a short sub-pass in the first pass to train, then uses adaptive
  - Pros: Adapts earlier, keeps overhead similar to adaptive
  - Cons: May be mis-trained by the small sub-pass



Background: Scheduling Policies: Quick

- Breaks the **first** pass into sub-passes to train
  - Uses a short sub
  - Pros: A
  -

Assumes each iteration does similar computation across passes!



GEMM: Accelerated OpenMP

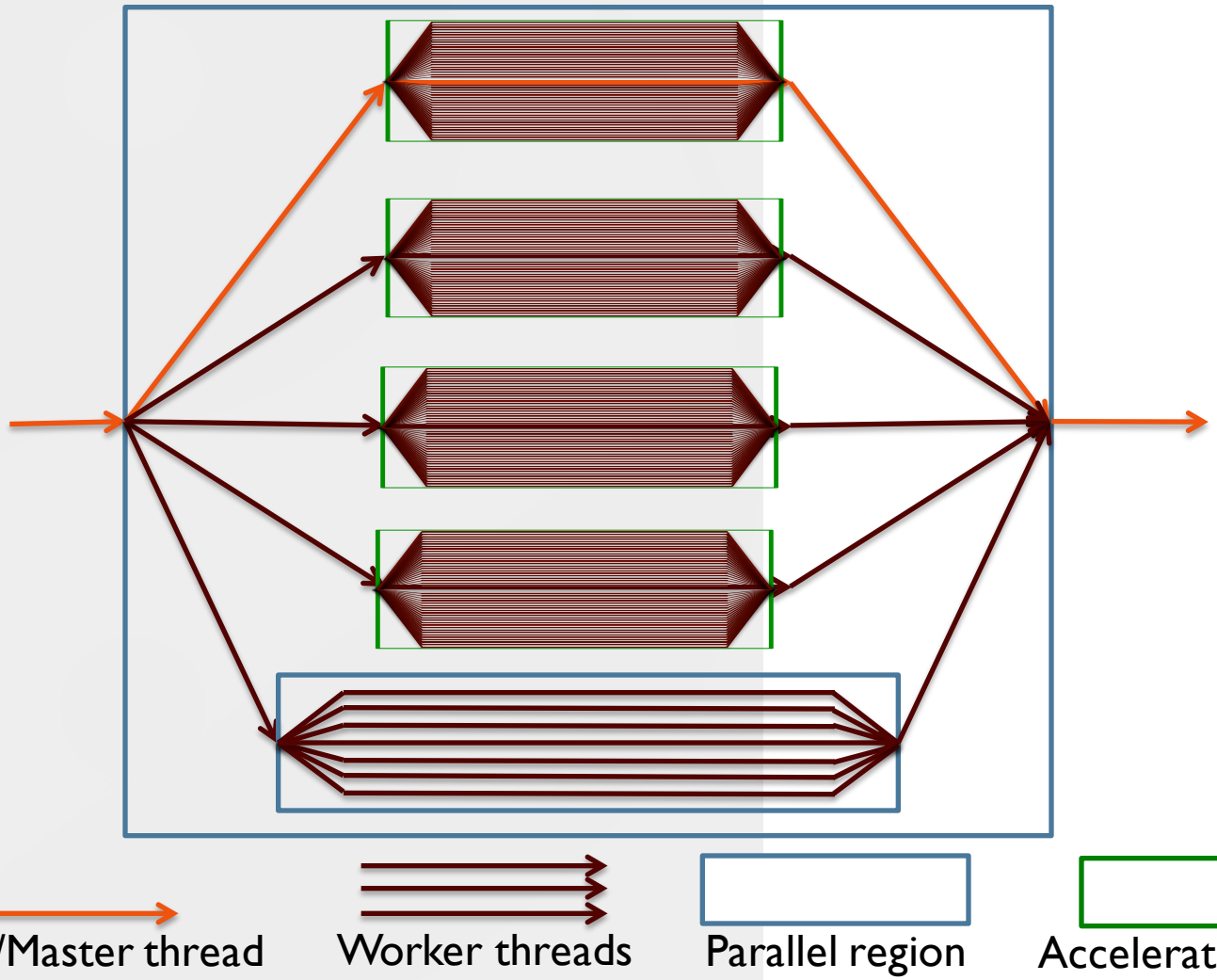
```
void runGemm(T *a, T *b, T *c) {
#pragma omp target teams distribute parallel for \
                map(tofrom: c[0:N][0:N])   \
        map(to: a[0:N][0:N], b[0:N][0:N])
for (int i = 0; i < N; i++) {
   for (int j = 0; j < N; ++j) {
     c[(i*N) + j] *= B;
     for (int k = 0; k < N; ++k) {
       c[(i*N) + j] += A * a[(i*N) + k] * b[(k*N) + j];
     }
   }
  }
}
```

GEMM: CoreTSAR Extended Accelerated OpenMP

```
void runGemm(T *a, T *b, T *c) {
#pragma omp target teams distribute parallel for  \
        map(partial, tofrom: c[true:N][false:N])  \
        map(partial, to: a[true:N][false:N])      \
        map(to: b[0:N][0:N])                      \
        hetero(true, all, adaptive)
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; ++j) {
      c[(i*N) + j] *= B;
      for (int k = 0; k < N; ++k) {
        c[(i*N) + j] += A * a[(i*N) + k] * b[(k*N) + j];
      }
    }
  }
}
```

Targets: **All** local accelerators **and** CPU cores

CoreTSAR Threading Behavior



Solutions

- Adaptive scheduling to minimize launch overhead (Splitter)
- Performance-prediction for load-balancing (CoreTSAR)
- Multi-device memory management (CoreTSAR)

Performance Prediction: Our Previous Approach

- Designed to "split" work between CPUs and one GPU
- Used a simple extrapolation to balance predicted runtime

Issue: Only works for *two* devices!

- Where:

i_t : Total iter...

p_g : Time/Iteration for GPU

p_c : Time/Iteration for CPUs

Performance Prediction: Take Two

- Integer optimization program
  - globally optimal distribution based on the prediction
  - Minimize difference in runtime between all devices

Issue: It is *slow*

$$\sum_{j=1}^{n-1} (t_j^+ + t_j^-)$$

I = tot

i_j = itera

f_j = fraction of

p_j = recent time/iteration for device j

n = number of devices

t_j^+ (or t_j^-) = time over (or under) equal

$$\sum_{j=1} = I$$

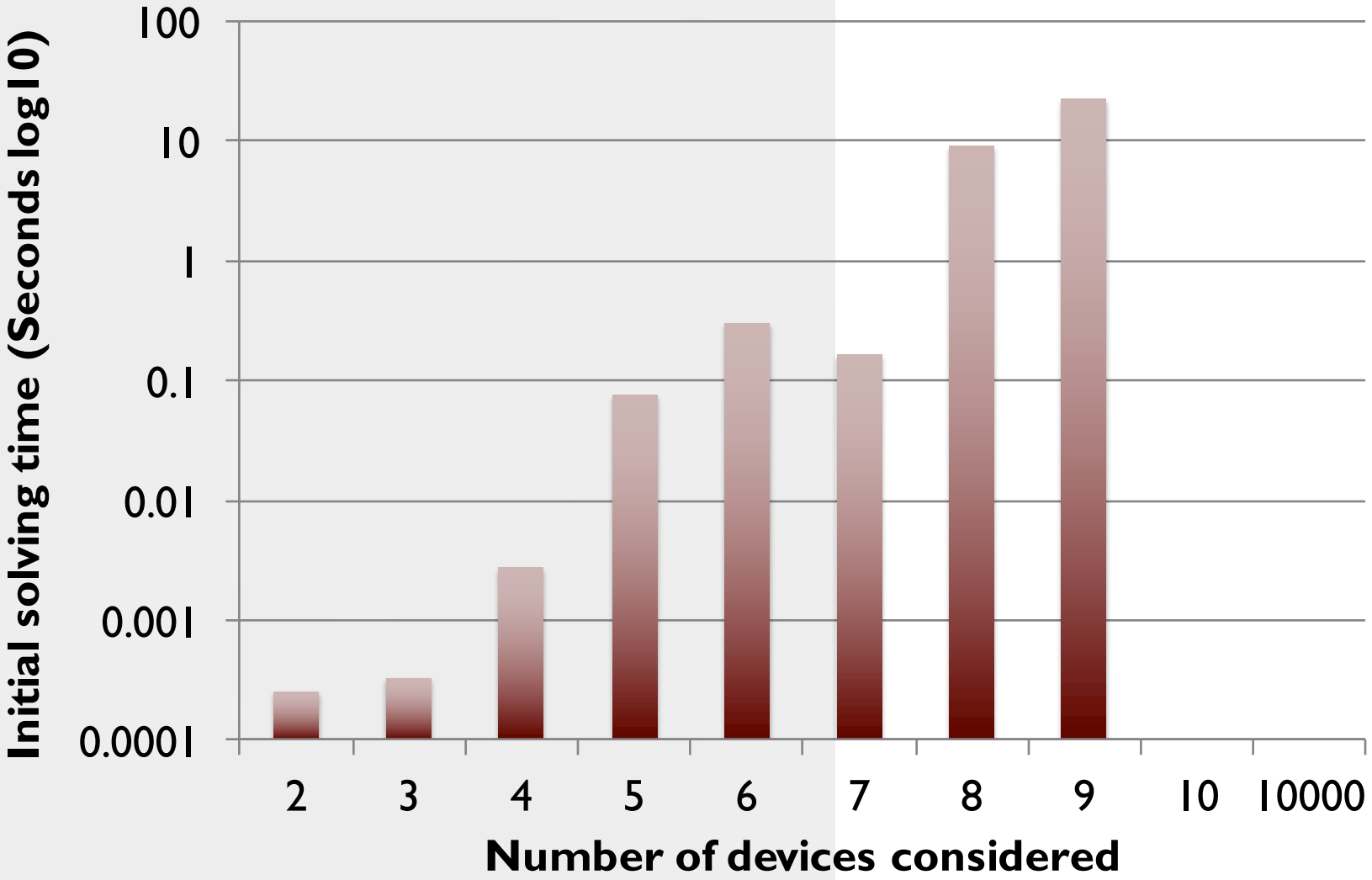
$$i_2 * p_2 - i_1 * p_1 = t_1^+ - t_1^-$$

$$i_3 * p_3 - i_1 * p_1 = t_2^+ - t_2^-$$

$$\vdots$$

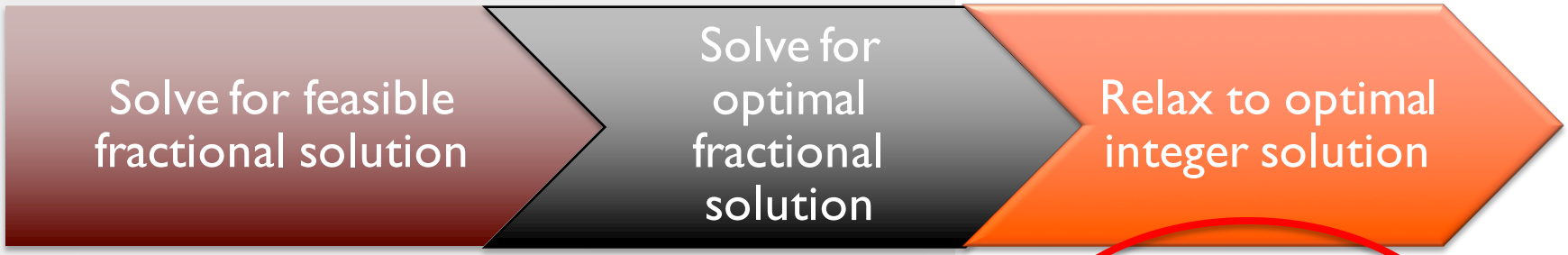
$$i_n * p_n - i_1 * p_1 = t_{n-1}^+ - t_{n-1}^-$$

Integer Linear Program Performance



What's Taking so Long?

For a solve with 8 devices



8 Iterations

8 Iterations

894,973 Iterations!

Performance Prediction: Take Three

- Linear optimization program
  - Solves for a **percentage** of iterations rather than total number
  - Guaranteed to be within [illegible] optimal on each device

How does it perform?

$I =$ tot

$i_j =$ iterat

$f_j =$ fraction of ite

$p_j =$ recent time/iteration for device j

$n =$ number of devices

t_j^+ (or t_j^-) = time over (or under) equal

$$t_j^-)$$

$$\sum_{j=0} j_j = 1$$

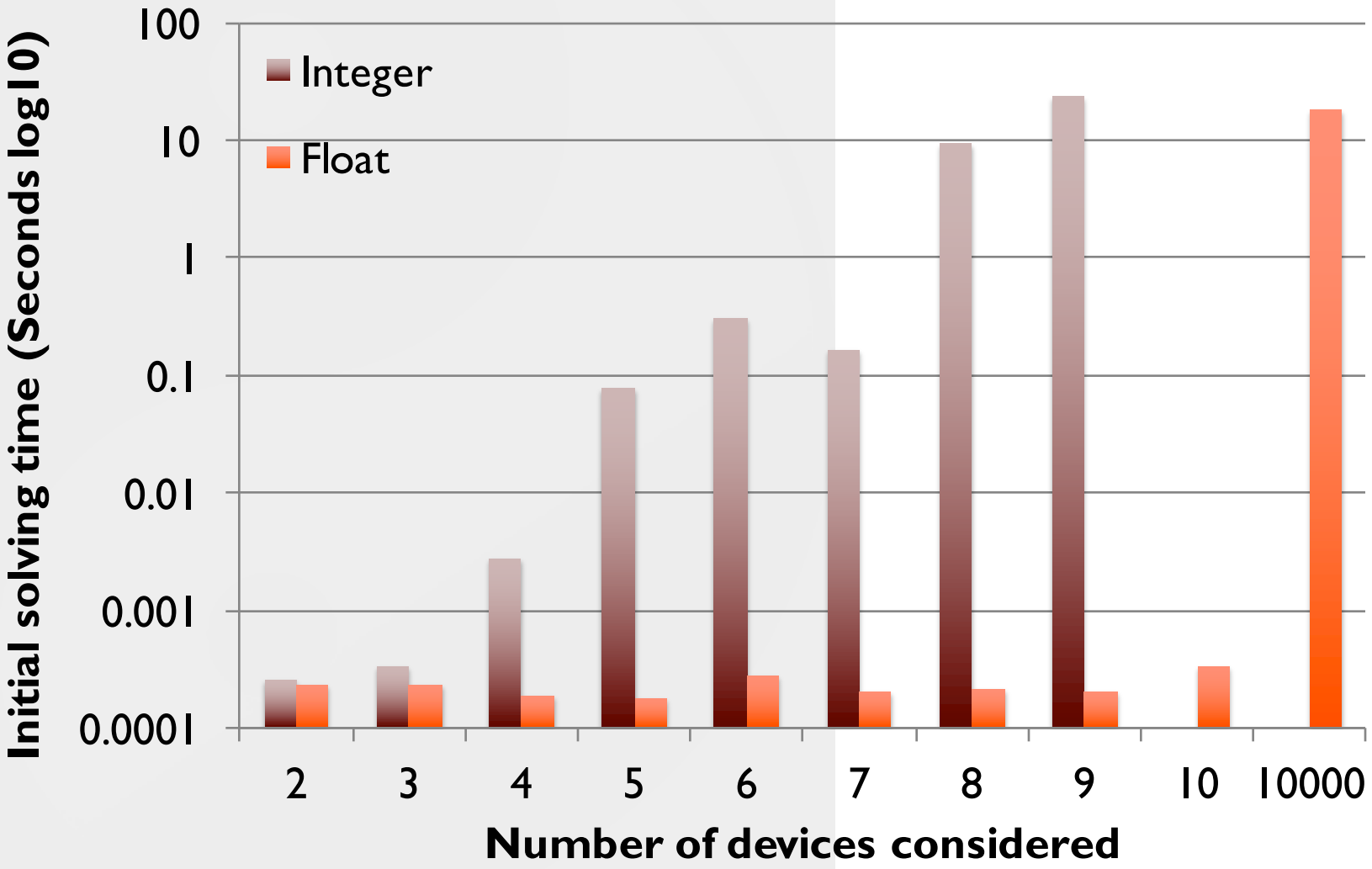
$$f_2 * p_2 - f_1 * p_1 = t_1^+ - t_1^-$$

$$f_3 * p_3 - f_1 * p_1 = t_2^+ - t_2^-$$

$$\vdots$$

$$f_n * p_n - f_1 * p_1 = t_{n-1}^+ - t_{n-1}^-$$

Linear Program Performance



Solutions

- Adaptive scheduling to minimize launch overhead (Splitter)
- Performance-prediction for load-balancing (CoreTSAR)
- Multi-device memory management (CoreTSAR)

Memory Association:

- Accelerated OpenMP:
  - Maps a logically-contiguous block of memory to a compute region
  - Can target only one device per thread at a time

- The alternative (or extension)
  - Specify association between iterations and data
  - Compute input/output set from assigned iterations in the runtime
  - Reduce memory transfers by only copying assigned data
  - Single directive is sufficient for any number of devices
  - Maintain consistency by merging **all** output into host memory at the end of each region

Memory Association Syntax

```
map(partial, <direction>: <variable>[<assoc.>:<length>:<boundary>])
```

- <variable>: An array, matrix or pointer that conforms to the requirements of the copy, copyin, or copyout clauses
- <assoc.>: Whether to associate the dimension with the accelerator
- <length>: The number of elements in the dimension
- <boundary>: Number of "boundary" elements required in this dimension

Memory Association Example: GEMM

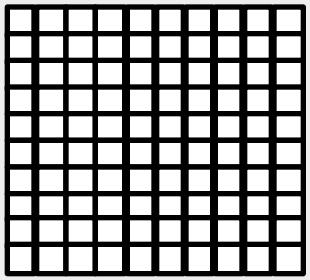
```
void runGemm(T *a, T *b, T *c) {
#pragma omp target teams distribute para
        map(partial, tofrom: c[true:N][fal
        map(partial, to: a[true:N][false:N]) \
        copyin(b[0:N][0:N])      \
        hetero(true, all, adaptive)
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; ++j) {
      c[(i*N) + j] *= B;
      for (int k = 0; k < N; ++k) {
        c[(i*N) + j] += A * a[(i*N) + k] * b[(k*N) + j];
      }
    }
  }
}
```

Rows, and not columns, associated with i

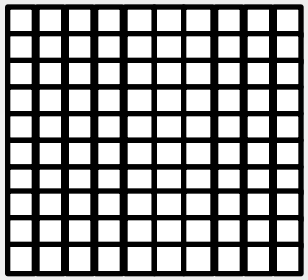
All of b is required on each device

Memory Association Example: What Actually Happens?

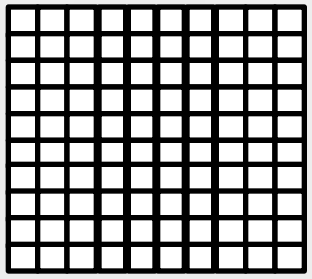
A: row associated

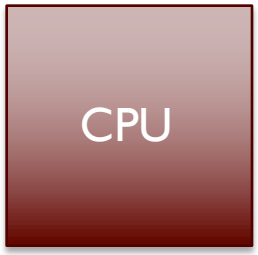


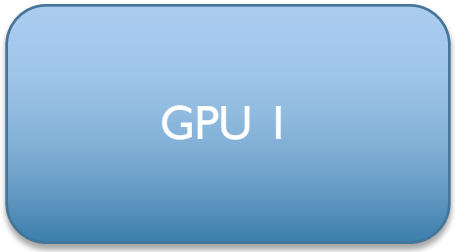
B: replicated



C: row associated

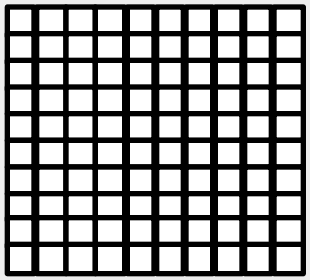




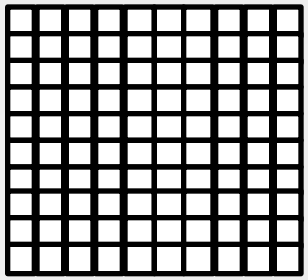


Memory Association Example: Scheduling

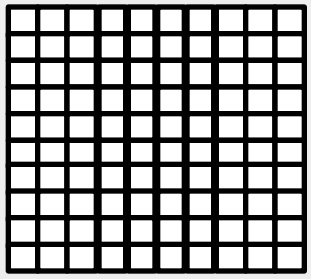
A: row associated

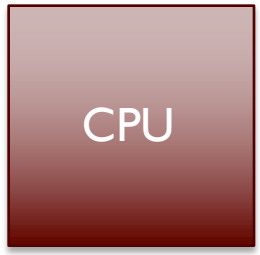


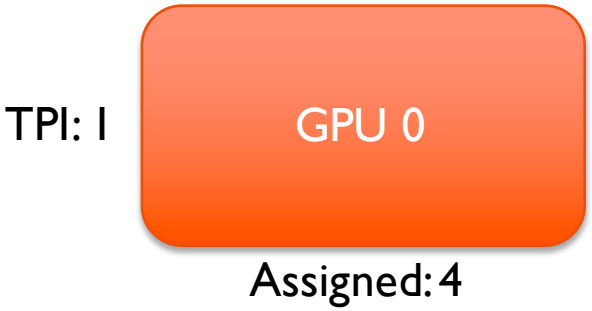
B: replicated

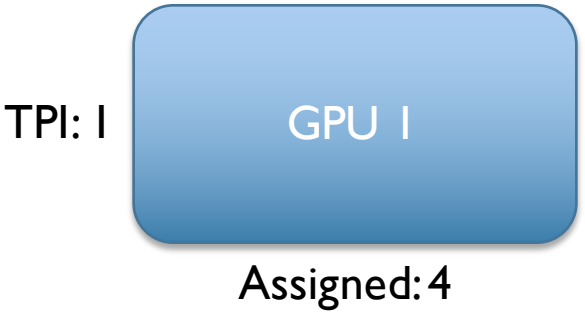


C: row associated

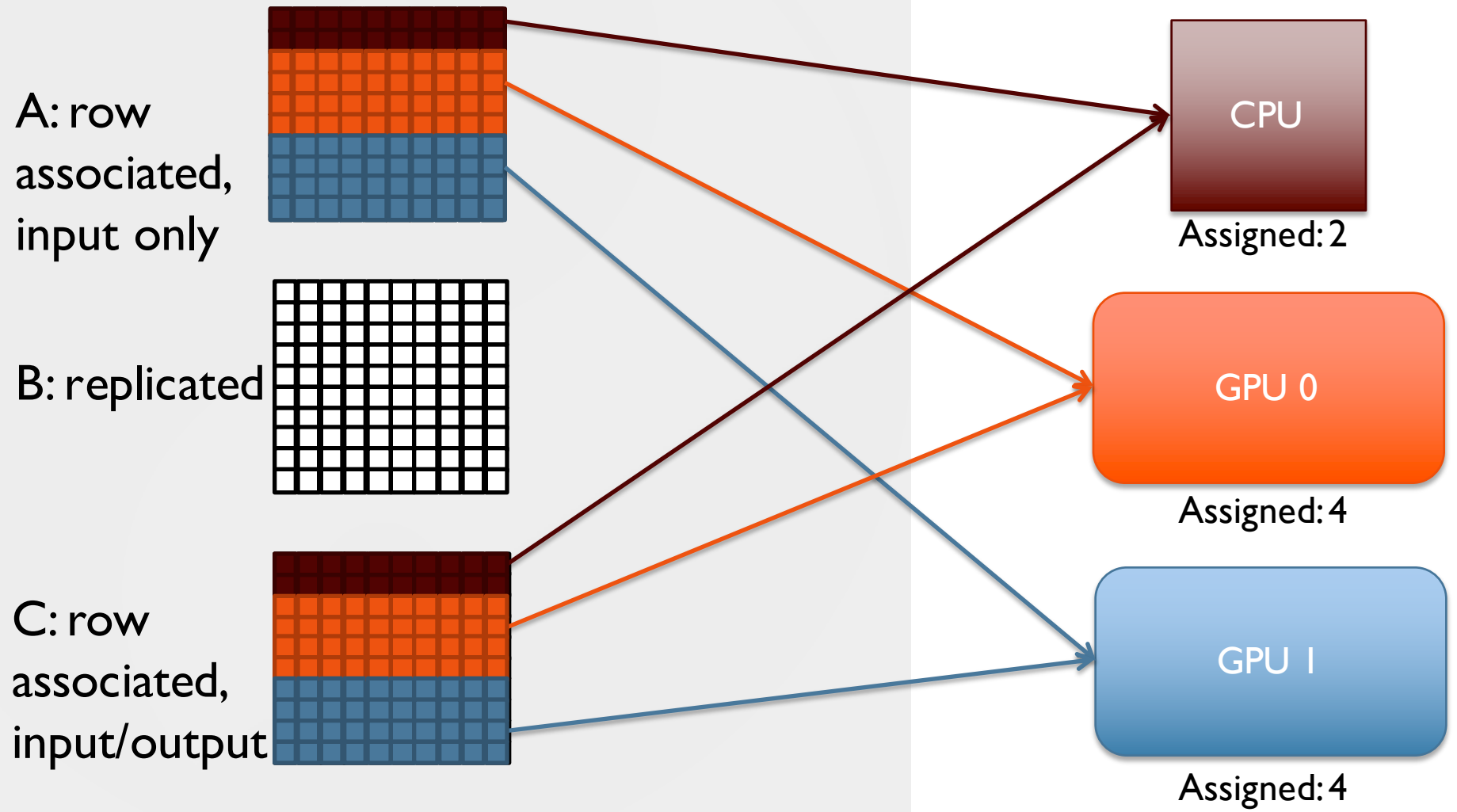






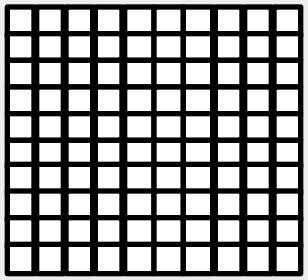


Memory Association Example: Data Distribution: Input

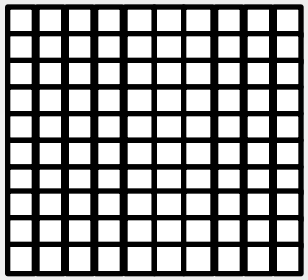


Memory Association Example: Data Distribution: Output

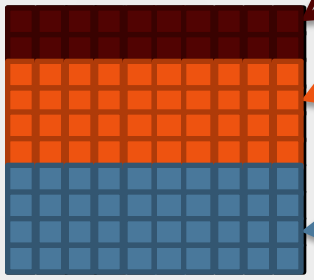
A: row associated, input only



B: replicated



C: row associated, input/output



Given the associations, **all** data movement is handled by CoreTSAR

CPU

Assigned: 2

GPU 0

Assigned: 4

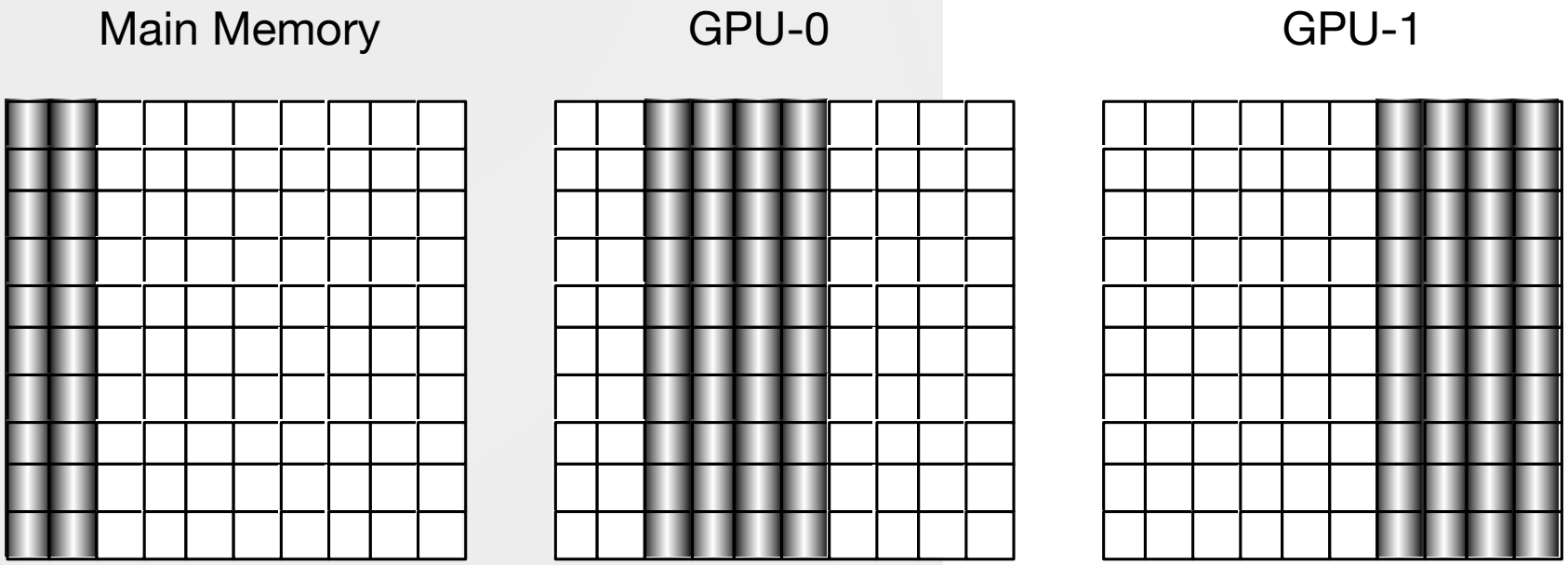
GPU 1

Assigned: 4

CoreTSAR Memory-Association Example: Simple Column-Wise Association

Directive:

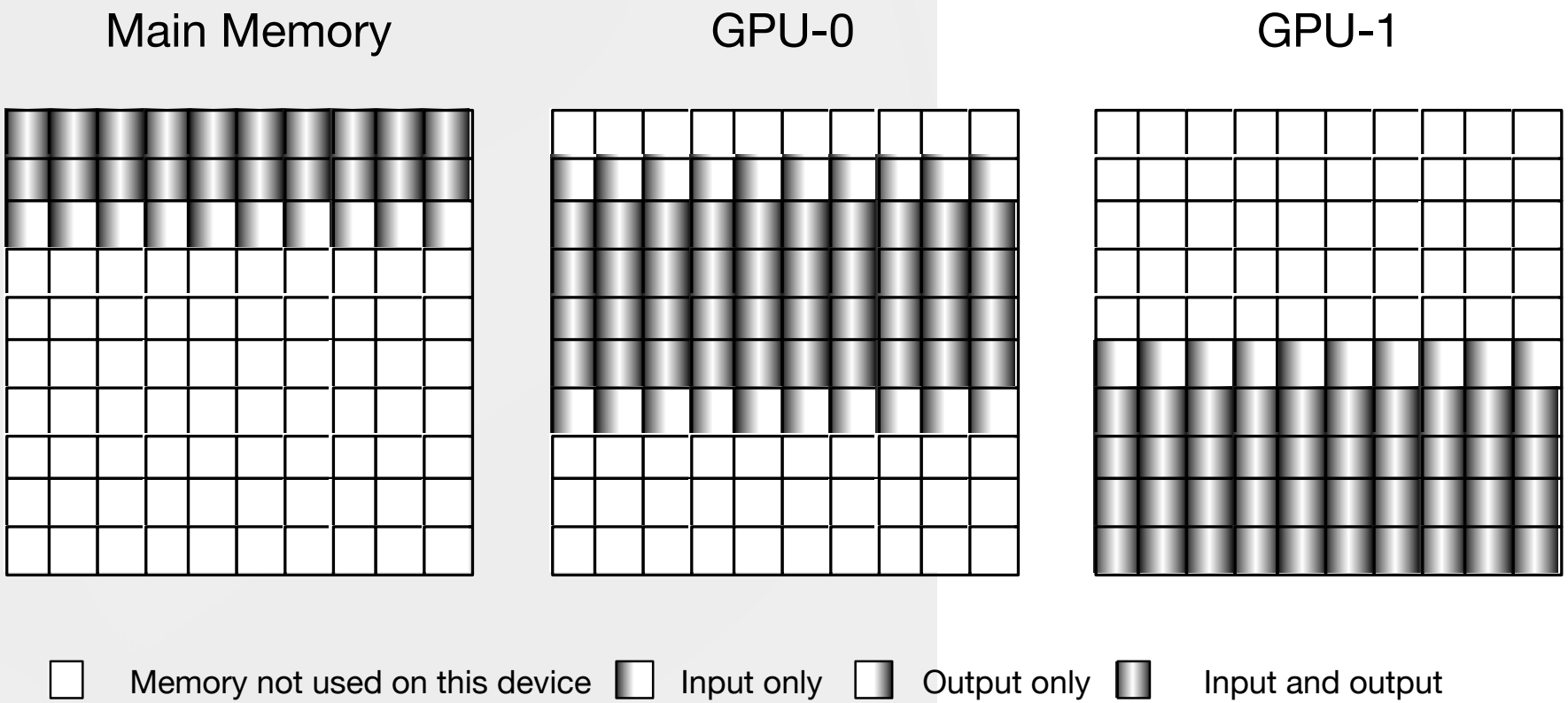
```
#pragma acc region hetero(TRUE) \
pcopy(mat[false:10][true:10])
```



CoreTSAR Memory-Association Example: Row-Wise Single-Boundary

Directive:

```
#pragma acc region hetero(TRUE) \
  pcopy(mat[true:10:1][false:10])
```



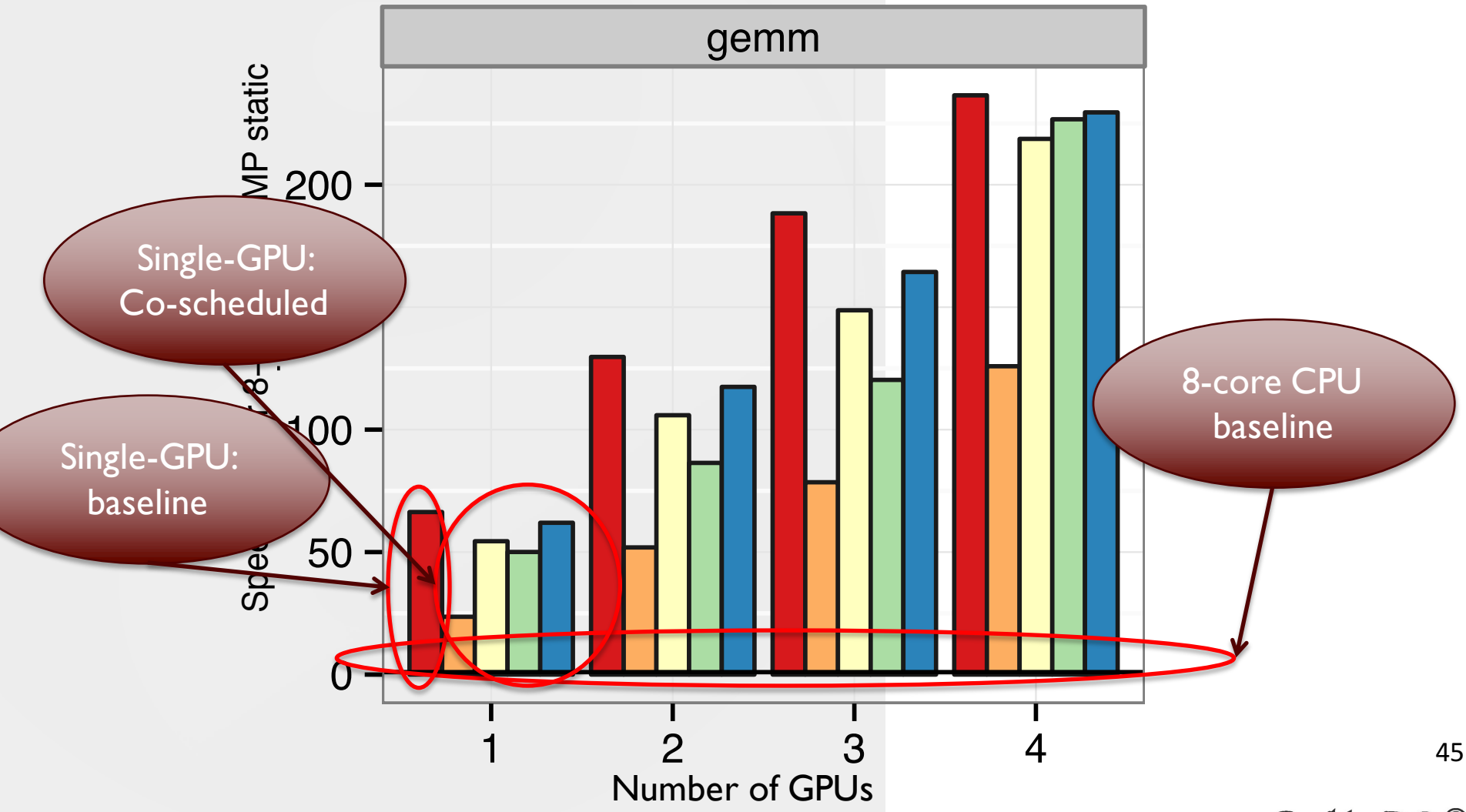
Results: Benchmarks
Representative Subset

- GEMM – Few passes
  - Matrix multiplication benchmark from PolyBench/GPU
- K-Means – Few passes
  - Iterative clustering of points
- Helmholtz – Few passes, GPU Unsuitable
  - Jacobi iterative method implementing the Helmholtz equation
- CORR – Few passes, heterogeneous iterations
  - Upper-triangular correlation matrix solver from PolyBench/GPU, all iterations do different amounts of work

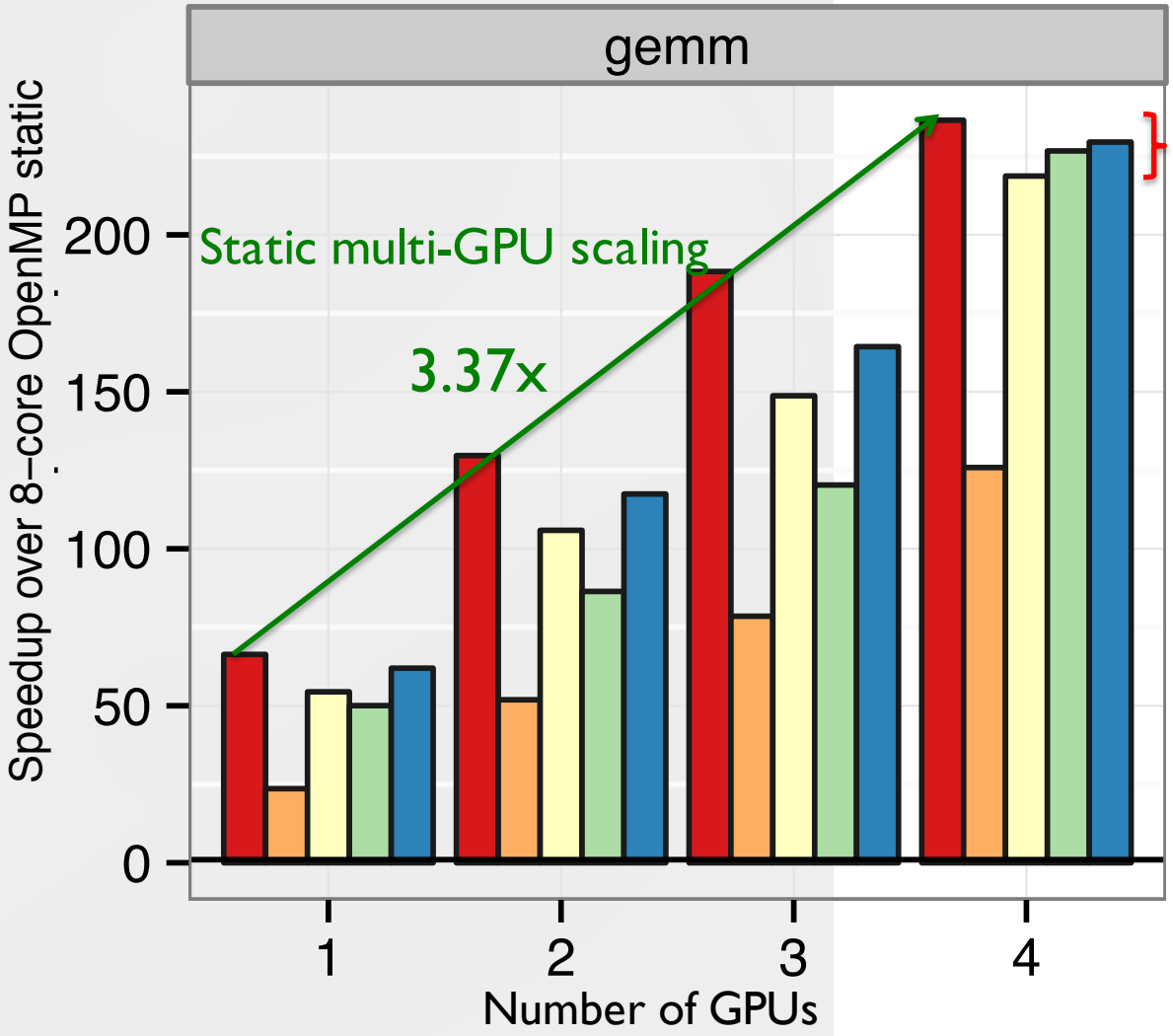
Results: Experimental Setup

- System
  - 2x 4-core Intel X5550 CPUs
  - 4x NVIDIA Tesla C2070 GPUs
  - Debian Squeeze Linux
  - PGI Accelerator compiler version 12.9
- Procedures
  - All parameters default, unless otherwise specified
  - Results represent 5 or more runs

Results for Co-Scheduling Amenable Benchmarks: GEMM: Dense Linear Algebra, Matrix Multiplication

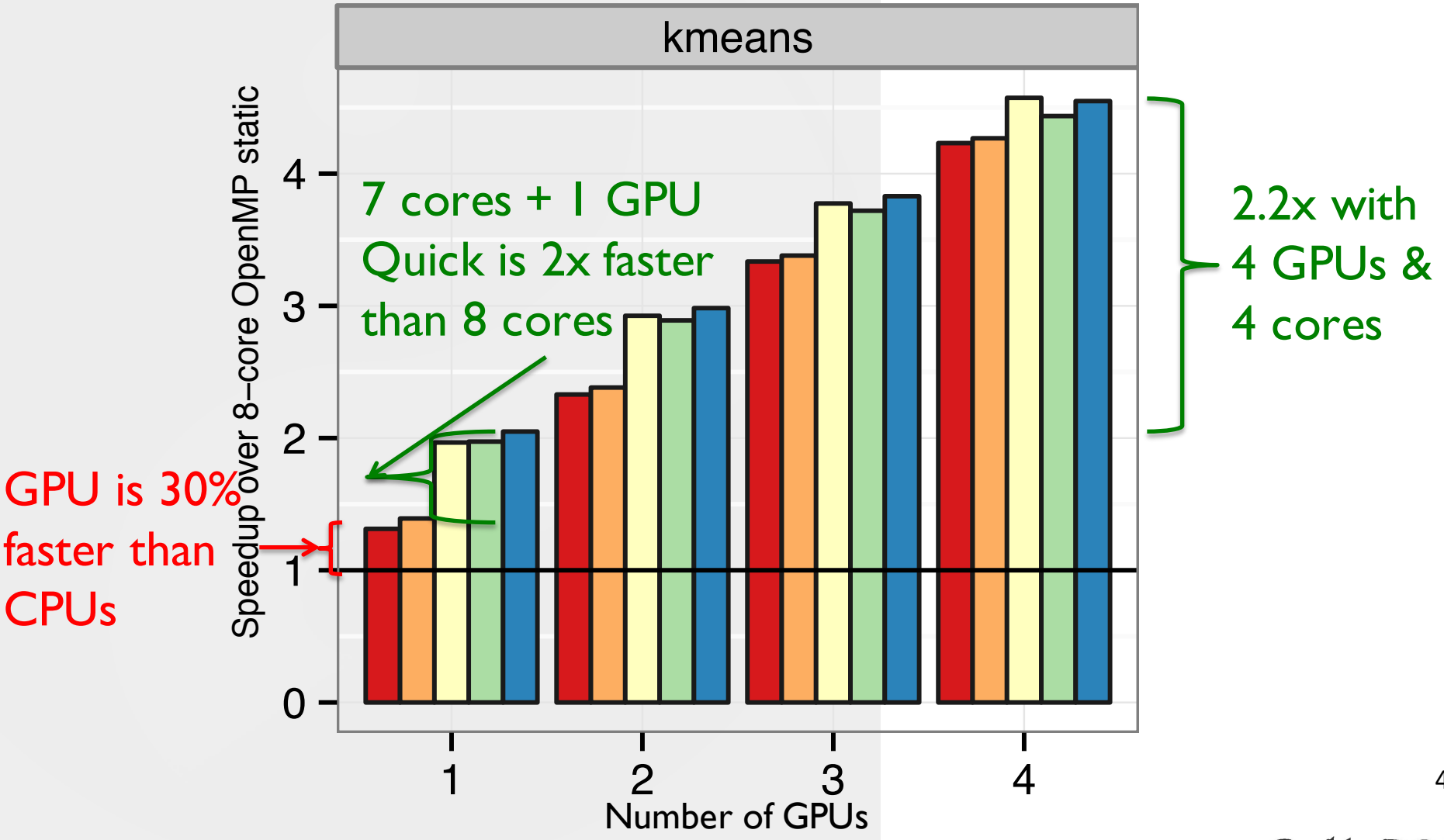


Results for Co-Scheduling Amenable Benchmarks: GEMM: Dense Linear Algebra, Matrix Multiplication

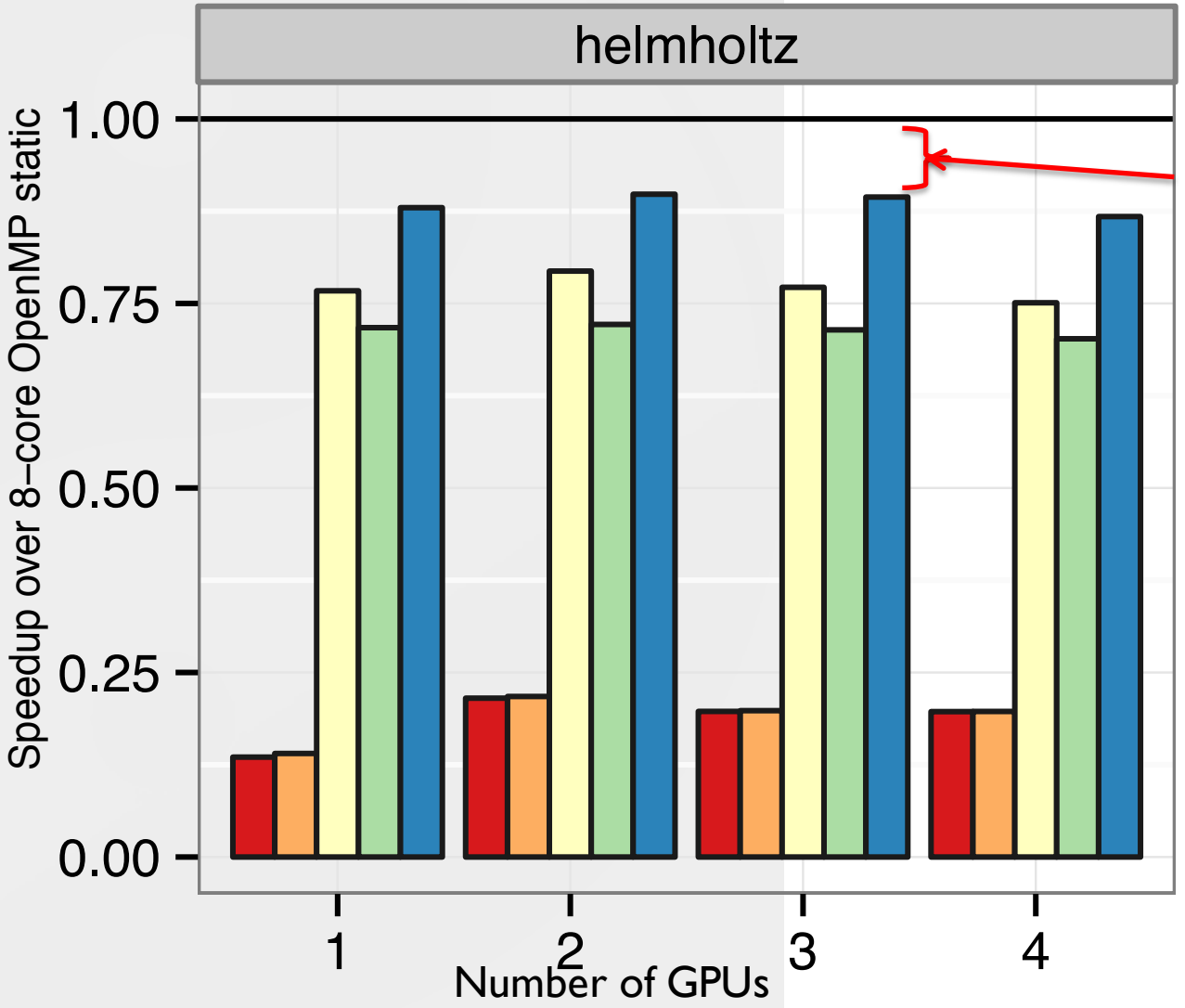


No gain from CPUs, **One** row can be too much for a core

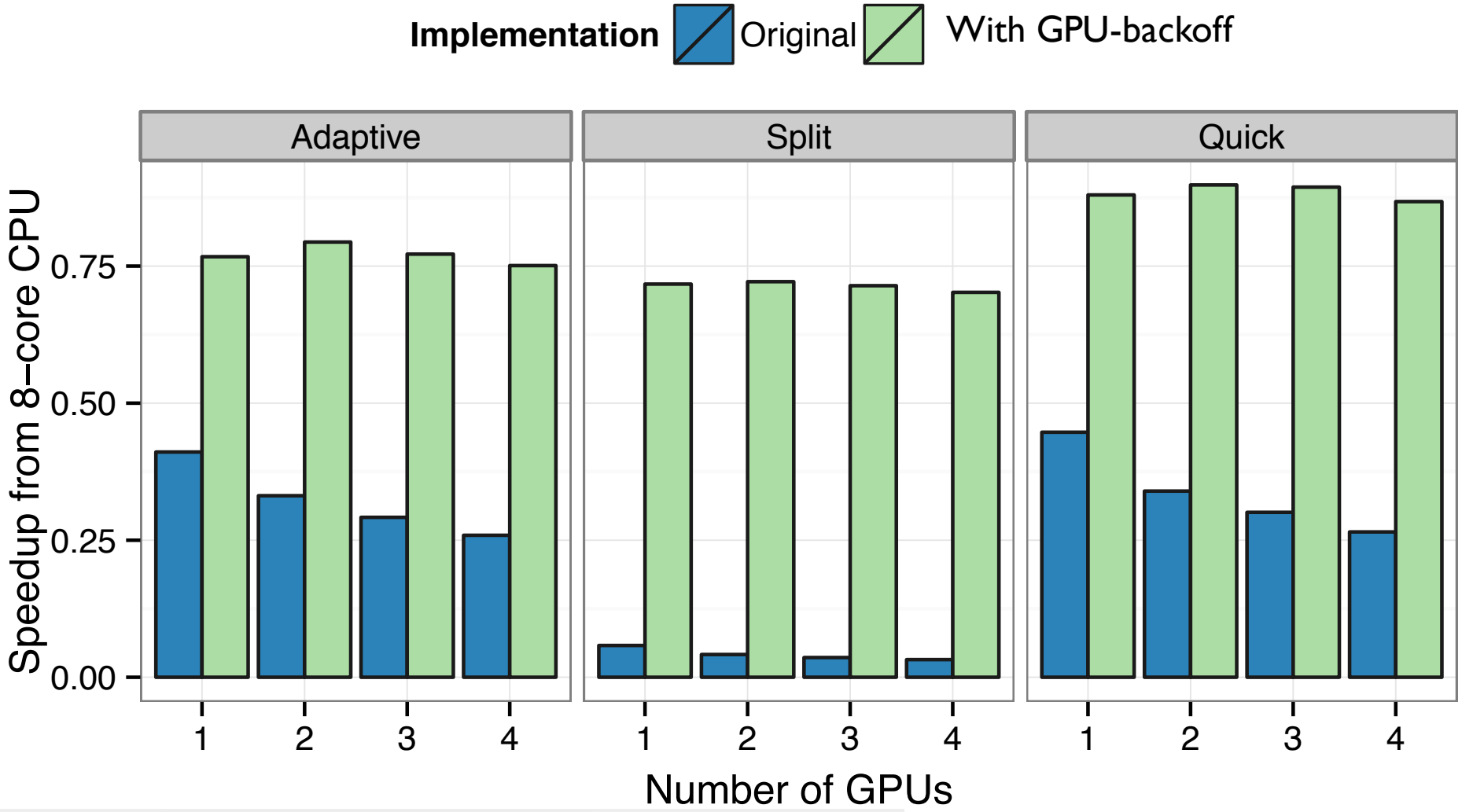
Results for Co-Scheduling Amenable Benchmarks: K-Means: Clustering Algorithm



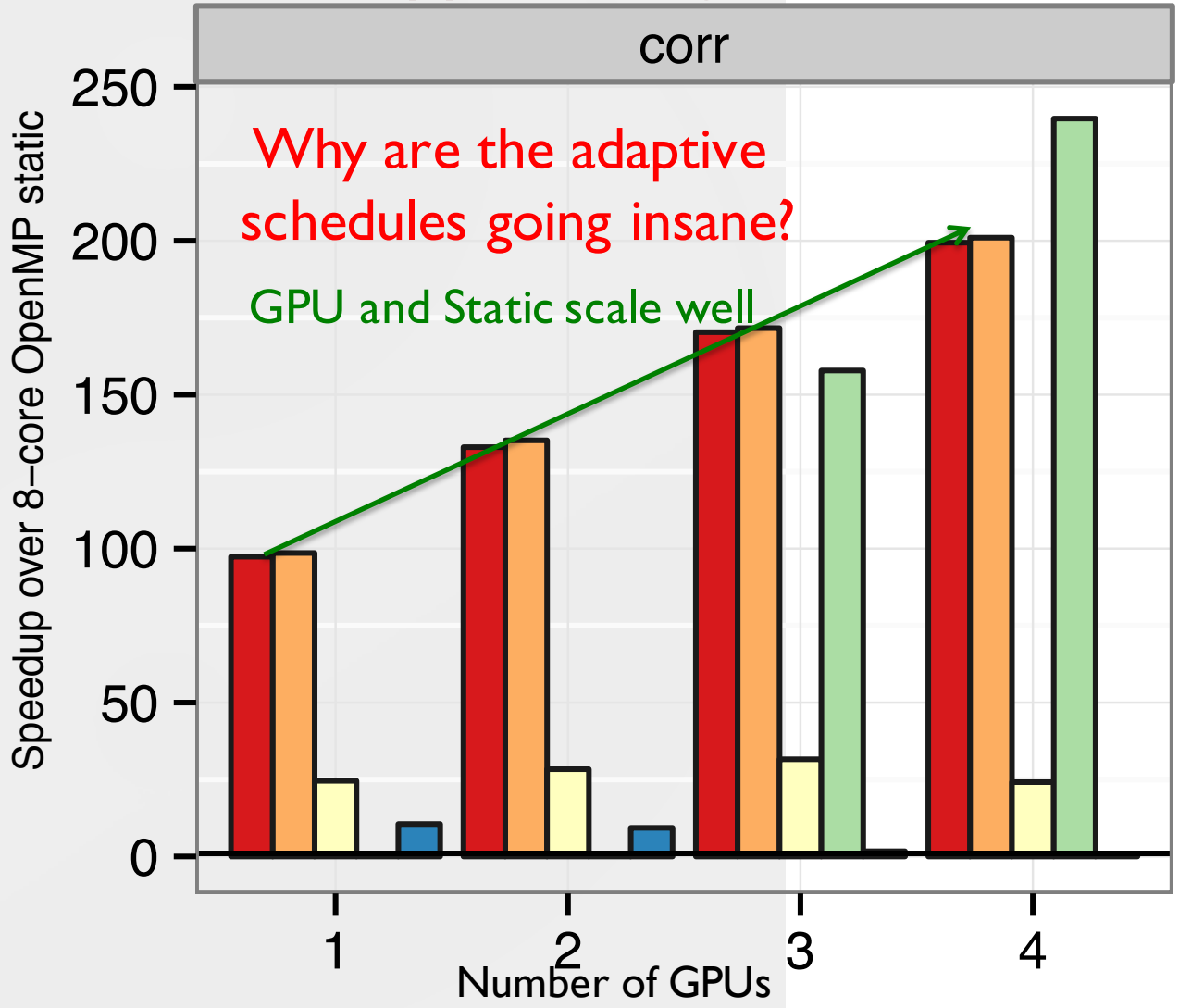
Results for Co-Scheduling Averse Benchmarks: Helmholtz: Jacobi Solver of the Helmholtz Equation



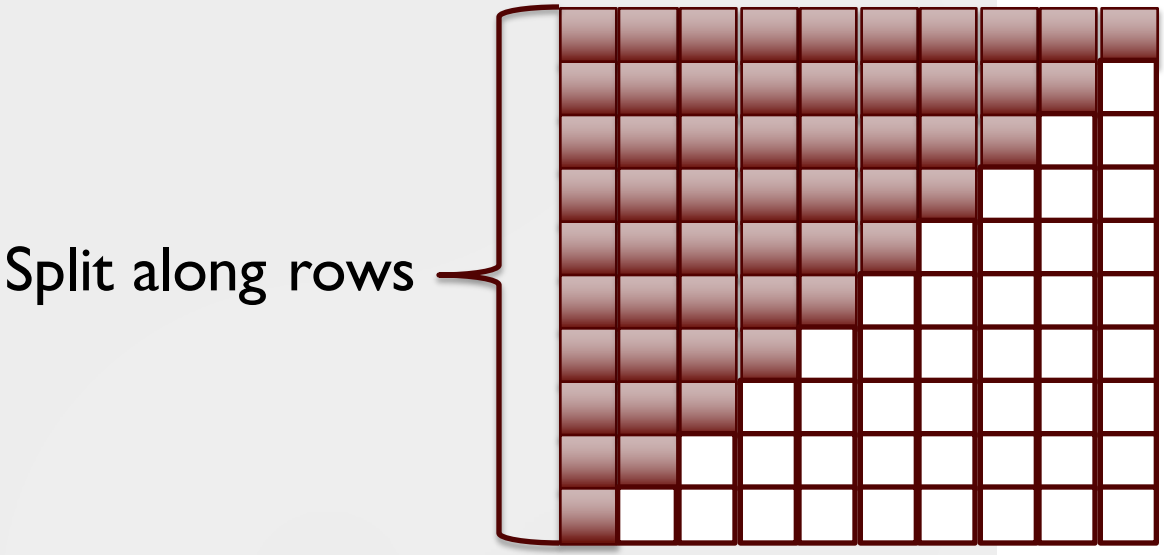
Results for Co-Scheduling Averse Benchmarks: Helmholtz: With and Without GPU Back-off



Results for Co-Scheduling Averse Benchmarks: PolyBench CORR: Upper Triangular Matrix, Correlation



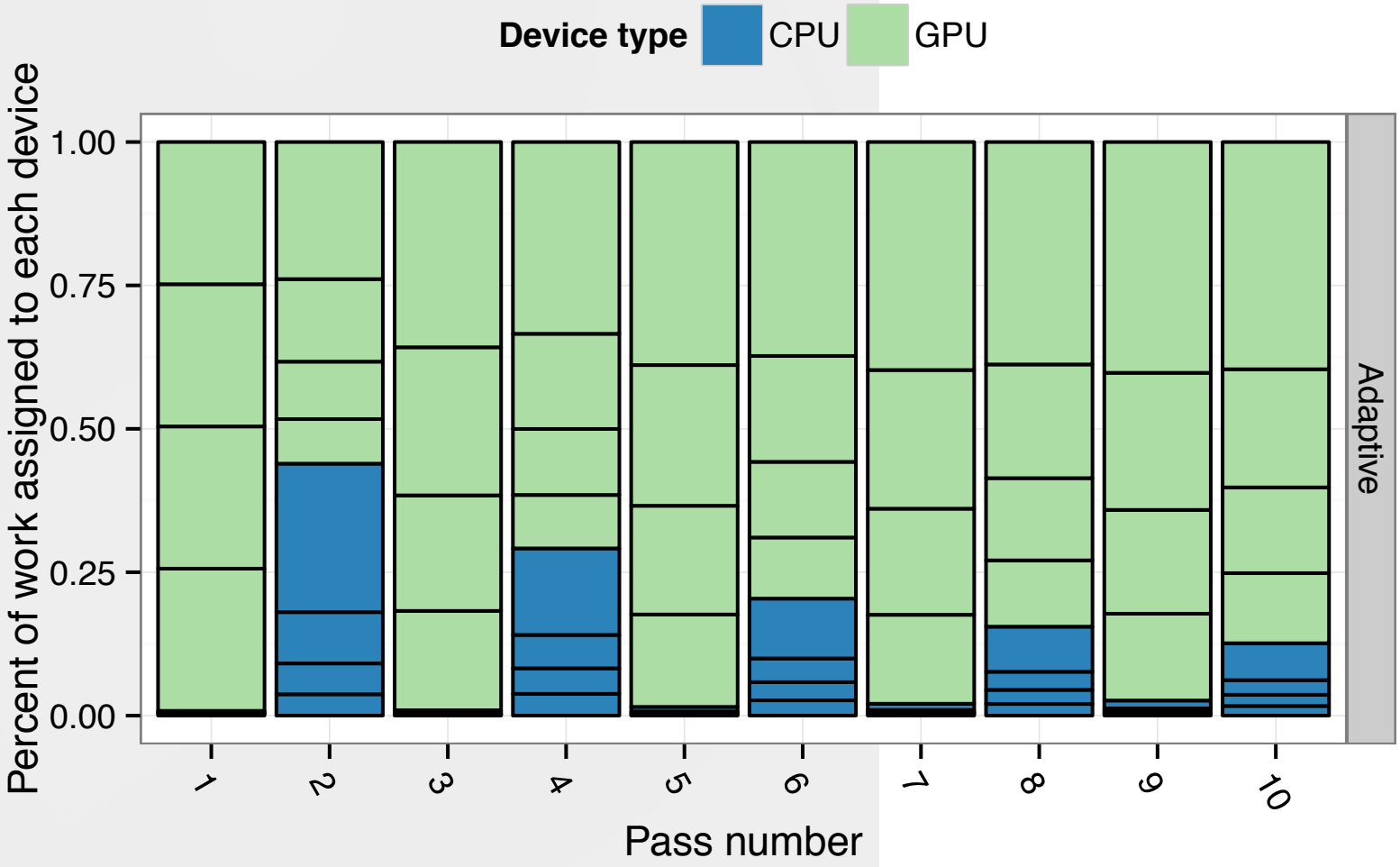
CORR Execution Pattern



Every iteration, and range, does a different amount of work!

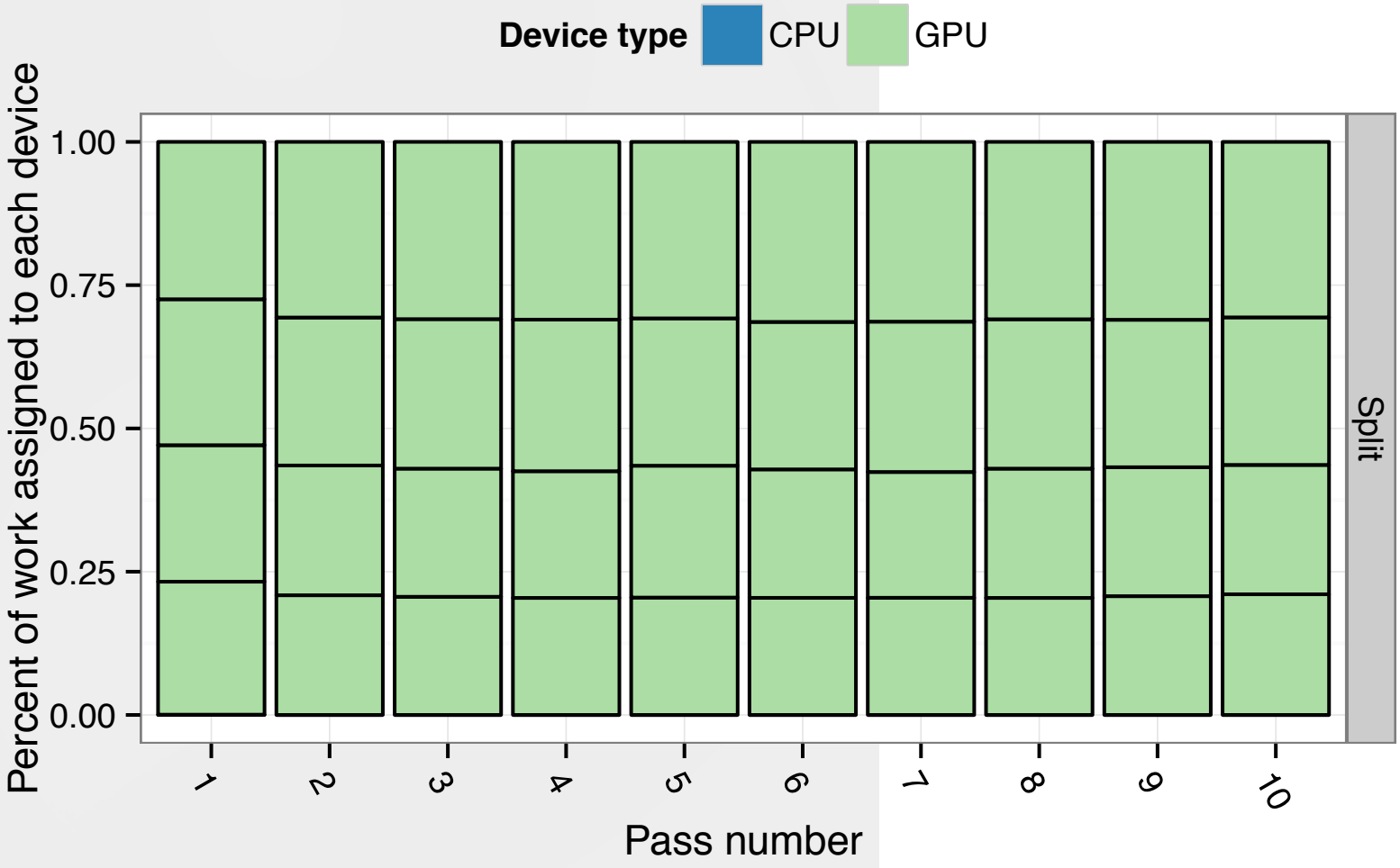
CORR Scheduling Behavior with Adaptive

Uneven work between iterations causes oscillations in Adaptive

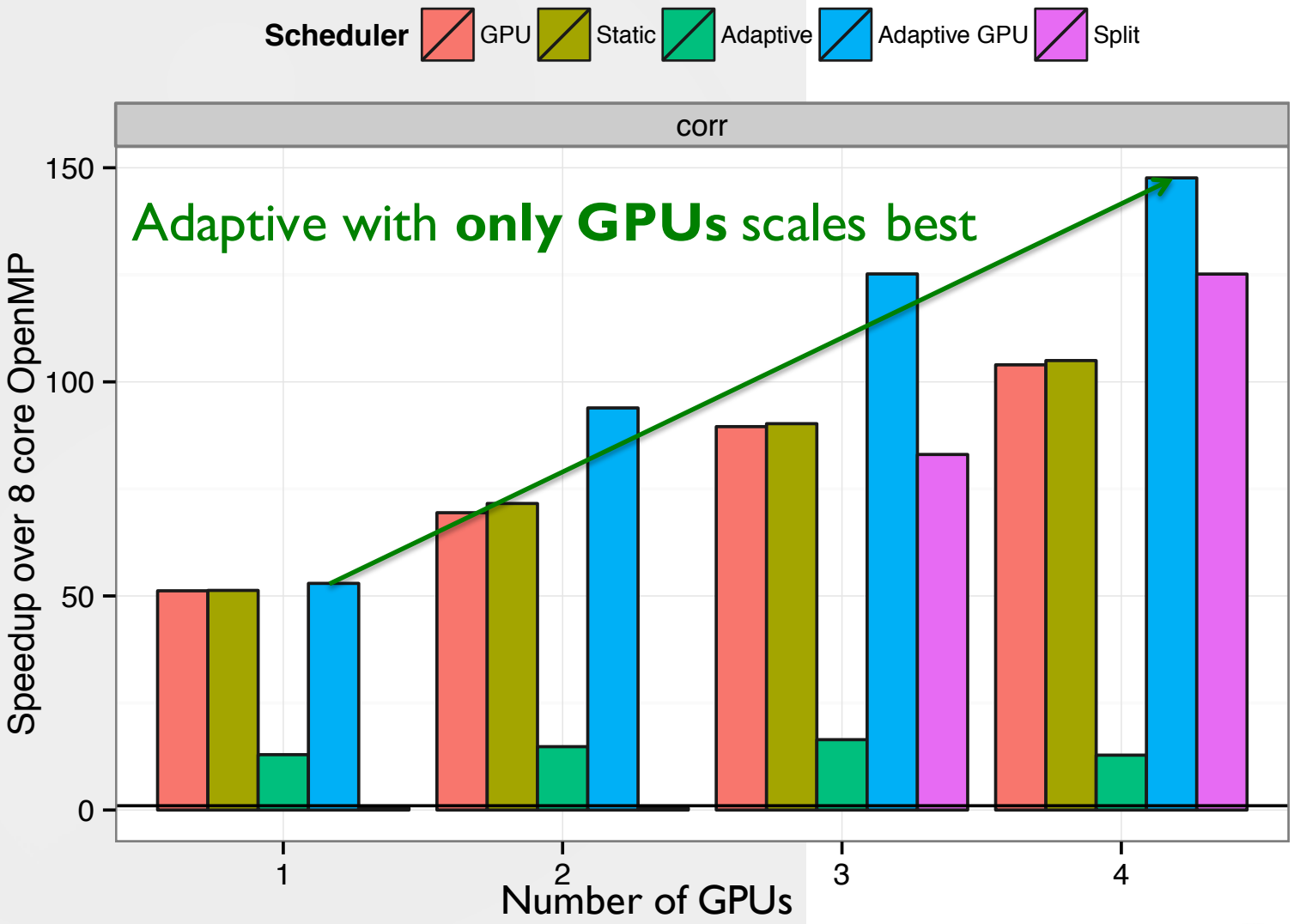


CORR Scheduling Behavior with Split

Split does better, by not using the CPU cores!



CORR with Adaptive GPU-only



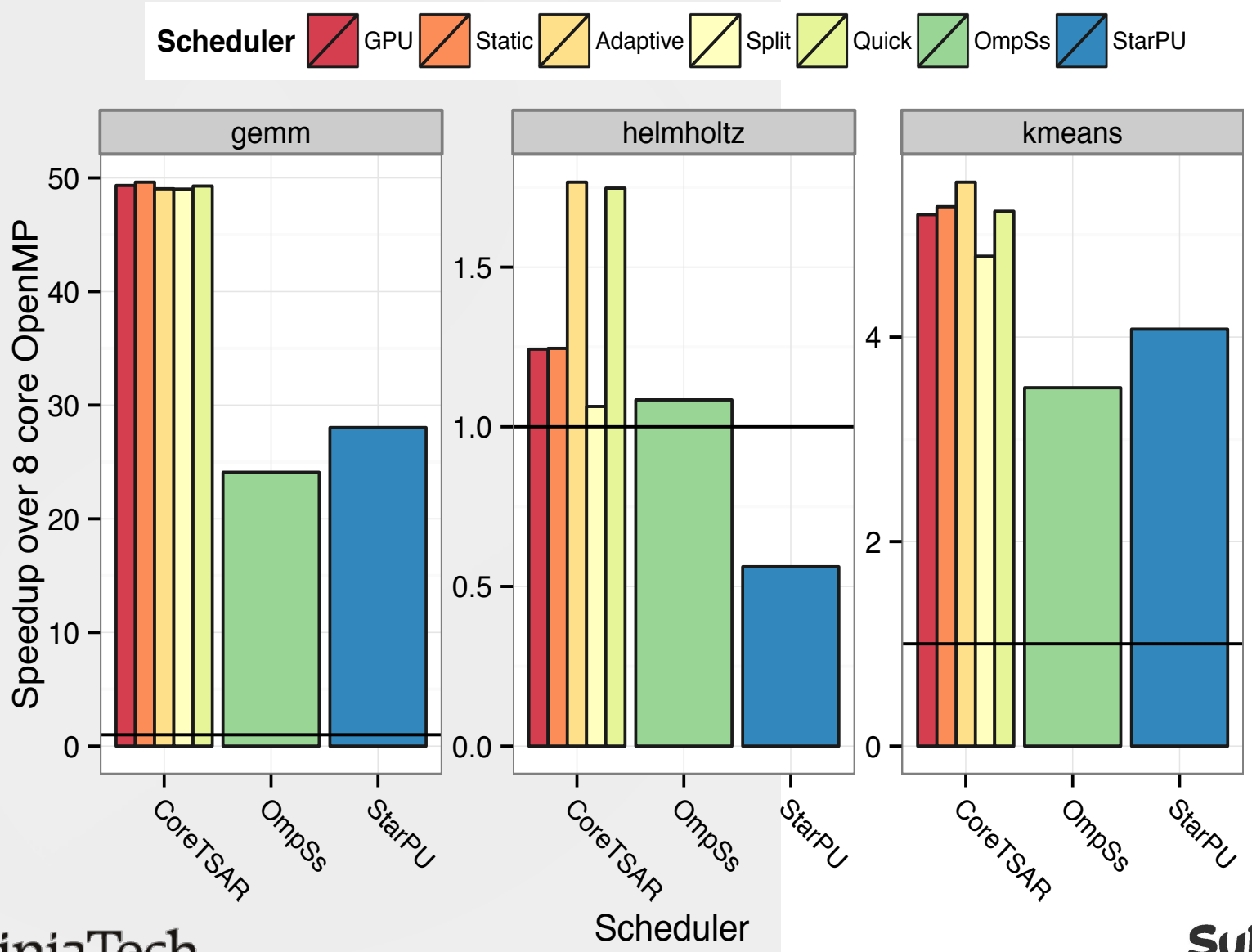
Comparison with State of the Art

- Chose two popular task schedulers
  - OmpSs
  - StarPU
- Both are **general *task* schedulers**
  - Arbitrary graphs of dependent tasks can be expressed
  - Scheduling is focused on distributing discrete *tasks* rather than loop iterations
- Our approach is complementary to theirs

Comparison with State of the Art: Setup

- Ported three of the benchmarks to the OmpSs and StarPU task schedulers
  - CUDA/c versions of each kernel function added
  - OmpSs: Versioning stack scheduler used to allow alternatives, automatic scheduling between CPUs and GPUs
  - StarPU: directly implemented in c API using the history based performance model, primed with 10 runs, and the dmda scheduler
- Created CoreTSAR scheduled versions using the CUDA/C kernels
  - CoreTSAR does ***not*** **require accelerated OpenMP regions**
  - Allows direct 1-1 comparison with identical compute regions
- All compiled with –O3 on:
  - nvcc: CoreTSAR and StarPU
  - mnvcxx: OmpSs

Results Compared with OmpSs and StarPU



Conclusions

- Adaptive scheduling and dynamic task granularity provided speedups of as much as 2x over alternative methods
- Our dynamic schedulers achieve high performance across various applications and highly heterogeneous machines
- Heterogeneous task scheduling for accelerated OpenMP is a feasible and useful extension

Questions?

Future Directions

- Explore other accelerators such as FPGAs
  - OpenCL vs Synthesis
- Cost of memory movement not included in model
  - Stencil codes need boundaries which are calculated in each iteration
  - Cost vs benefit of moving boundaries between accelerators in each iteration
  - Asynchronous coherence?
- Uneven work in each iteration across passes
  - Application profiling?